www.icgst.com

Performance Evaluation of Integrated Fault-Tolerant Technique: Simulation Study

O. A. Abulnaja*, N. M. Saadi**
* *King Abdulaziz University, Jeddah, Saudi Arabia*
e-mail: abulnaja@kaau.edu.sa
** *Faculty of Technology, Jeddah, Saudi Arabia*
e-mail: nms178@hotmail.com

Abstract
In earlier work we have proposed the concept of the *dynamic group maximum matching* for grouping the system graph into groups of different sizes according to the tasks arriving at the system. Also, we have developed a more efficient integrated fault-tolerant technique for ultra-reliable execution of tasks where both hardware (processors and communication channels) and software failures, and *on-line fault diagnosis* are considered. The proposed approach called the *Integrated Fault-Tolerant* (*IFT*) approach. Furthermore, we have proposed integrated fault-tolerant scheduling algorithms. The introduced algorithms are based on the dynamic group maximum matching concept and the IFT technique.

In this work, we studied the effect of the IFT technique on system performance for four of the proposed scheduling algorithms. The algorithms are: *Integrated Fault-Tolerant First-Come, First-Served* (*FCFS*), *Integrated Fault-Tolerant* (*FCFS + Smallest Fits First*) (*FCFSSFF*) scheduling algorithm, *Integrated Fault-Tolerant* (*FCFS + Largest Fits Firs*t) (*FCFSLFF*) scheduling algorithm, and *Integrated Fault-Tolerant* (*FCFS + First Fits First*) (*FCFSFFF)* scheduling algorithm. We considered two performance metrics: *system mean response time* and *percentage of completed tasks of specific type*.

Keywords: *Performance Evaluation; Fault Tolerance; Fault Diagnosis; Task Scheduling; Networks; Systems Reliability.*

1. Introduction
Various studies have shown that both hardware and software are subject to failures. However, the majority of the existing works have dealt with the problem by considering that either software is fault-free but hardware is subject to failure, for instance see [1] - [4], or hardware is fault-free but software is subject to failure, for instance see [5] - [11]. Thus, techniques for dealing with hardware and software faults (integrated techniques) must be developed.

In an earlier work [12] we have introduced a more efficient new integrated fault-tolerant technique called the Integrated Fault-Tolerant (IFT) technique, where both hardware (processors and communication channels) and software are subject to failures. The proposed technique has the capability of on-line fault diagnosis. In the following subsections we discuss the work.

1.1. Dynamic Group Maximum Matching Concept
In an earlier work [12] we have introduced the concept of the dynamic group maximum matching for grouping the system graph into groups of different sizes according to the tasks arriving at the system. We have also proposed the *Dynamic Group Maximum Matching (DGMM)* algorithm for finding the dynamic group maximum matching.

The maximum number of hardware faults that a system can tolerate with respect to a task T_i is defined as the task *hardware reliability degree* t_{hi}. As a task hardware reliability degree increases, more redundancy is used. In [13, 14], the researchers assumed that all the tasks running in the system have equal hardware reliability degree t, and they partitioned the system into groups of size $(t + 1)$.

The concept of *group maximum matching* has been introduced by Hosseini in [13], which is a generalization of the classical maximum matching concept. The concept of the classical matching problem is used to group nodes of a graph into 2-node disjoint groups. A generalization to the classical matching is to group the nodes into $(t + 1)$-node disjoint groups. In classical maximum matching problem, 2-node nodes are grouped such that the number of groups is maximum. Similarly, the generalization maximum matching problem, nodes are grouped (each group is of size $(t + 1)$) such that the number of groups is maximum. Also, the researcher has proposed the *Group Maximum Matching* (*GMM*) algorithm for finding the group

maximum matching. In [14], the researchers have shown that the GMM algorithm most of the time generates a maximum number of groups and rarely generates one group less than the maximum number. One drawback of the group maximum matching concept relates to the system performance where the system resources may not be utilized efficiently because less critical tasks (tasks with hardware reliability degree $t_{hi} < t$) will use more resources than what they need to maintain their reliability requirements. A second drawback of the concept relates to the system reliability. If a small t is used, then tasks with higher reliability requirements will run with lower hardware reliability degree.

In [12], we have introduced the dynamic group maximum matching concept, which is introduced to overcome the above shortcoming and is a generalization of the group maximum matching concept. In this generalization, the system is partitioned into disjoint groups with different sizes dynamically. When a task T_i with the hardware reliability degree t_{hi} is scheduled by the scheduler for execution, a group of processors of size $g_i = t_{hi} + 1$ is assigned to the task. We also have proposed the Dynamic Group Maximum Matching (DGMM) algorithm for finding the dynamic group maximum matching. The proposed algorithm is a greedy heuristic algorithm and attempts to avoid the isolation of the system processors and attempts to include them in groups. This is achieved by including the processors with lower degrees in groups first and then the processors with higher degrees. At the same time the DGMM algorithm attempts to minimize the time needed to release the correct outputs and maximize the on-line faults diagnoses capabilities. This is achieved by trying to increase the group *connectivity*. For example, consider a task T_i with the hardware reliability degree $t_{hi} = 2$. If we can allocate the task T_i to a fully connected group of 5 processors with 2 faulty processors, we will get the correct output and at the same time we can diagnose the faulty processors in the group upon the execution of the task by the processors. However, if we allocate the task T_i to a linear array group of 5 processors, that may not be possible. The formal description of the DGMM algorithm and an illustrating example are given in [12].

1.2. Integrated Fault-Tolerant (IFT) Technique

The ultimate goals for any computer system design are: reliable execution of tasks (high reliability) and on-time delivery of service (high performance). Thus, the ultimate goal is to concurrently optimize reliability and system performance, while noting how achieving each of the above goals separately affects the other one. For example, increasing software reliability means using more redundant software, thereby lowering system performance. Another example relates to fault diagnosis where there is a need to either run periodically diagnostic programs or use redundant resources. In both cases the system performance is degraded due to the extra time overhead spent to diagnose the faults.

Our approach considers the viability of achieving the above goals simultaneously. In our work an attempt is made to maximize the system reliability and the system performance while concurrently diagnosing both hardware and software faults. In the following we will outline the proposed work:

- *High Reliability Approach*: Our work considers the system as a whole, an integration of hardware and software. Here, both hardware failures and software failures are considered in contrast to the most of the existing works that have assumed that only one of them, not both, could be faulty.
- *High Performance Approach*: In contrast to most of the existing works that have focused mainly on improving the system reliability and have used system resources lavishly, we attempt to maximize the performance concurrently. The following list some of our concerns:
    1. Since every system is fault-free most of the time, allocating a task T_i to $(2t_{hi} + 1)$ processors to tolerate t_{hi} hardware faults, as is done in some of the existing works, is a waste of the system resources. Instead, we allocate initially $(t_{hi} + 1)$ processors to the task T_i, which is minimal for tolerating t_{hi} hardware faults, and in case of failures we add more processors as needed. A similar procedure is used for tolerating software failures. It is important to realize that software is fault-free most of the time as well.
    2. In earlier work we have proposed the Dynamic Group Maximum Matching (DGMM) algorithm for grouping the system graph. The DGMM algorithm always attempts to maximize the system performance by increasing the number of concurrent tasks in the system.
    3. On-Line Fault Diagnosis: In or work faults will be diagnosed by running user programs, in contrast to some of the existing works that require running diagnostic programs. By implementing an on-line fault diagnosis, the system will be continuously executing useful application programs instead of executing diagnostic programs for failure detection which add extra overhead and may not providing 100% fault coverage.

The Integrated Fault-Tolerant (IFT) technique is devised for reliable task execution and on-line fault diagnosis, where processors, communication

channels, and software (application tasks) are subject to failures. For reliable execution of tasks, different program versions of each task are assigned to a group of processors. Processors are grouped using the DGMM algorithm. A task is released if at least (t_{hi} + 1) processors agree with each other on the outputs for at least (t_{si} + 1) different program versions and the outputs of all the program versions are the same, where t_{hi} denotes the upper bound for the number of faulty processors and communication channels (hardware reliability degree) and t_{si} denotes the upper bound for the number of faulty program versions (software reliability degree) that the system can tolerate with respect to a task T_i.

1.2.1. Comparison Model of Computation

In the Integrated Fault-Tolerant (IFT) approach, the comparison model works as follows. When two neighboring processors P_i and P_j finish executing a program version V_{kl} of a task T_l, they exchange and compare their outputs and then the processor P_i obtains its test outcomes for the assigned task as follows:

1. For every program version V_{ml} of the task T_l executed by the processor P_i prior to the program version V_{kl} do
   (a) If the output of P_i for the program version V_{kl} agrees with the output of P_i for the program version V_{ml} then
      i. a_{ii} (V_{kl}, V_{ml}) = 0.
   (b) Else
      i. a_{ii} (V_{kl}, V_{ml}) = 1.
2. For every program version V_{ml} of the task T_l executed by the neighboring processor P_j so far do
   (a) If the output of P_i for the program version V_{kl} agrees with the output of P_j for the program version V_{ml} then
      i. a_{ij} (V_{kl}, V_{ml}) = 0.
   (b) Else
      i. a_{ij} (V_{kl}, V_{ml}) = 1.

The neighboring processor P_j will follow similar steps to obtain its test outcomes.

Remarks:

1. a_{ij} and a_{ji} may not be the same.
2. 1. A faulty processor, a faulty channel, a faulty program version or any combination of them could be the source of the fault.
3. Processors P_i and P_j may produce the same output and agree with each other on the output for a program version, even if one (or both) of them are faulty or executing a faulty program version, if the fault does not affect the output. For example, a faulty register within a processor does not affect the output if it is not used in executing the program version.

1.2.2. Disagreement Graph

A disagreement graph DG_i (N_i, E_i), where N_i is the set of nodes of DG_i and E_i is the set of edges of DG_i, with respect to a task T_i is obtained as follows. Every node $X \in N_i$ contains some processors of the group G_i and some program versions of the task T_i, such that for every processor $Pj \in X$ and processor $P_k \in X$, Pj and P_k agree with each other on the outputs for at least one program version (the same or different) of the task T_i. An edge exists between nodes $X \in N_i$ and $Y \in N_i$ if there exists a disagreement between a processor $Pj \in X$ and a processor $P_k \in Y$ on the outputs for at least one program version (the same or different) of the task T_i and the processor Pj and the processor P_k are neighbors or $j = k$ in the system G. Agreement operation has a *transitivity* property. That is if P_i and P_j agree with each other on the output for a program version V_{li} of the task T_i and in turn P_j and P_k agree with each other on the output for the program version V_{li} of the task T_i, then P_i and P_k agree with each other on the output for the program version V_{li} of the task T_i.

An illustrating example of the IFT technique is given in [12].

1.3. Integrated Fault-Tolerant Scheduling Algorithms

In [12], we have introduced several integrated fault-tolerant scheduling algorithms. These scheduling algorithms are based on the Integrated Fault-Tolerant (IFT) technique and the Dynamic Group Maximum Matching (DGMM) algorithm.

In this paper, due to the space limitations, we will limit our study of the effect of the IFT technique on system performance for only four scheduling algorithms and we will consider the rest of these scheduling algorithms in a follow up paper. Two performance metrics; system mean response time and percentage of completed tasks of specific type, will be evaluated for each one of the scheduling algorithms.

1.3.1. Integrated Fault-Tolerant First Come, First served (FCFS) Scheduling Algorithm

The Integrated Fault-Tolerant First-Come, First-Served (FCFS) scheduling algorithm work as follows. As tasks which may consist of more than one program version arrive at the system, they are queued up along with their group sizes (i.e, g_i = task hardware reliability degree t_{hi} + 1) and task software reliability degree t_{si} in a single task queue Q. When a task T_i is scheduled for execution, the DGMM algorithm is called to find the required group size for the task T_i. If the returned group size is equal to

the required group size, the first program version V_{1i} of the task T_i is assigned to the group G_i for execution; otherwise, the DGMM algorithm is called to find another subgraph of size g_i in a different part of the system graph. This process is repeated until either a group of size g_i is obtained or the entire system graph is searched without success. In the former case, the first program version V_{1i} of the task T_i is assigned to the group G_i for execution. In the latter case, the DGMM algorithm is called each time a task leaves the system or is inserted in the aborted task queue Q_a , to find a group of the required size. If the returned subgraph for execution; otherwise, the DGMM algorithm is called to find a new subgraph is equal to the required size in a different part of he system graph. This process is repeated until either a group of the required size is obtained or the entire system graph is searched without success. In the former case, the first program version of the task is assigned to all the processors in the group for execution. In the latter case, the above process is repeated until either the required group size is obtained or all the tasks left the system and the task T_i still cannot find the required group size. In the former case, the first program version of the task is assigned to all the processors in the group for execution. In the latter case, the task is aborted. When a program version V_{ji} of a task T_i completes its execution by all the processors in the group G_i, neighboring processors exchange and compare their outputs. Then, the disagreement graph DG_i is obtained. A task T_i is released if at least (t_{hi} + 1) different processors agree with each other on the output for at least (t_{si} +1) different program versions and the output for all the program versions are the same; otherwise, if there are at least (t_{hi} + 1) processors which agree with each other on the outputs for (2 t_{si}) or fewer different program versions and there are two or more different outputs, then the next program version of the task T_i is assigned to the group G_i for execution. Otherwise, if there are at least (t_{hi} + 1) processors which agree with each other on the outputs of (t_{si} +1) different program versions and there are three or more different outputs, then the task *Ti* is aborted; otherwise, the task group size is incremented by one ($gi = gi + 1$) and the DGMM algorithm is called to add one more neighboring processor to the processors in the group *Gi*. If the returned subgraph is equal to the required group size, the first program version V_{1i} of the task T_i is assigned to the group *Gi* for execution; otherwise, the DGMM algorithm is called to find another subgraph equal to the task group size in a different part of the system graph. Calling the DGMM algorithm is repeated until either a group of size *gi* is obtained or the entire system graph is searched without success. In later case, the task T_i is aborted and added to the aborted task queue Q_a for later execution. In the former case, the first program version is assigned to all the processors in the group for execution. The above process is repeated until the task is aborted, the task is aborted and inserted at the tail of the aborted task queue Q_a , or the output of the task is released. The formal algorithm is given in [12].

1.3.2. Integrated Fault-Tolerant (FCFS + Smallest Fits First) Scheduling Algorithm

The Integrated Fault-Tolerant (FCFS + Smallest Fits First) (FCFSSFF) scheduling algorithm works as follows. As tasks, which may consist of more than one program version, arrive at the system, they are queued up along with their group sizes (i.e., g_i = task hardware reliability degree t_{hi} + 1) and task software reliability degree t_{si} in a single task queue Q. When a task T_i is scheduled for execution, the Dynamic Group Maximum Matching (DGMM) algorithm is called to find the required group size for the task T_i. If the returned group size by the DGMM algorithm is smaller than the required group size, then the returned group is allocated to the first program version V_{1j} of the task T_j which has the smallest group size among the tasks in the task queue provided that the group size of the task T_j is not larger than the size of the returned group. Next, the DGMM algorithm is called to find another subgraph of size g_i in a different part of the system graph to allocate the task T_i. This process is repeated until either a group of size g_i is obtained or the entire system graph is searched without success. In the latter case, the task T_i is added to the aborted task queue Q_a for later execution. In the former case, the first program version V_{1i} of the task is T_i assigned to the returned group for execution. When a program version V_{ji} of a task T_i completes its execution by all the processors of its group G_i, neighboring processors exchange and compare their outputs. Then, the disagreement graph DG_i is obtained. A task T_i is released if at least (t_{hi} + 1) different processors agree with each other on the output for at least (t_{si} + 1) different program versions and the outputs of all the program versions are the same; otherwise, if there are at least (t_{hi} + 1) processors agree with each other on the output for ($2t_{si}$) or fewer different program versions and there are two or more different outputs, then the next program version of the task T_i is assigned to the group G_i for execution. Otherwise, if there are at least (t_{hi} + 1) processors which agree with each other on the outputs of ($2t_{si}$ + 1) different program versions and there are three or more different outputs, then the task T_i is aborted; otherwise, the task group size is incremented by one ($g_i = g_i + 1$), and the DGMM algorithm is called to add one more neighboring processor to the group G_i. Calling the DGMM algorithm is repeated until either a group of the required size is obtained or the entire graph is searched without success. In the latter case, the task T_i is aborted and added to the aborted task queue Q_a for later execution. In the former case, the first program version of the task is assigned to the returned group for execution. The above process is repeated until the task is aborted, the task is aborted and added to the aborted task queue

Q_a for later execution, or the output for the task is obtained. The formal algorithm is given [12].

1.3.3. Integrated Fault-Tolerant (FCFS + Largest Fits First) Scheduling Algorithm

The Integrated Fault-Tolerant (FCFS + Largest Fits First) (FCFSLFF) scheduling algorithm works as follows. As tasks, which may consist of more than one program version, arrive at the system, they are queued up along with their group sizes (i.e., g_i = task hardware reliability degree t_{hi} + 1) and task software reliability degree t_{si} in a single task queue Q. When a task T_i is scheduled for execution, the Dynamic Group Maximum Matching (DGMM) algorithm is called to find the required group size for the task T_i. If the returned group size by the DGMM algorithm is smaller than the required group size, then the returned group is allocated to the first program version V_{1j} of the task T_j which has the largest group size among the tasks in the task queue provided that the group size of the task T_j is not larger than the size of the returned group. Next, the DGMM algorithm is called to find another subgraph of size g_i in a different part of the system graph to allocate the task T_i. This process is repeated until either a group of size g_i is obtained or the entire system graph is searched without success. In the latter case, the task T_i is added to the aborted task queue Q_a for later execution. In the former case, the first program version V_{1i} of the task T_i is assigned to the returned group for execution. When a program version V_{ji} of a task T_i completes its execution by all the processors of its group G_i, neighboring processors exchange and compare their outputs. Then, the disagreement graph DG_i is obtained. A task T_i is released if at least (t_{hi} + 1) different processors agree with each other on the output for at least (t_{si} + 1) different program versions and the outputs of all the program versions are the same; otherwise, if there are at least (t_{hi} + 1) processors agree with each other on the output for ($2t_{si}$) or fewer different program versions and there are two or more different outputs, then the next program version of the task T_i is assigned to the group G_i for execution. Otherwise, if there are at least (t_{hi} + 1) processors which agree with each other on the outputs of ($2t_{si}$ + 1) different program versions and there are three or more different outputs, then the task T_i is aborted; otherwise, the task group size is incremented by one ($g_i = g_i + 1$), and the DGMM algorithm is called to add one more neighboring processor to the group G_i. Calling the DGMM algorithm is repeated until either a group of the required size is obtained or the entire graph is searched without success. In the latter case, the task T_i is aborted and added to the aborted task queue Q_a for later execution. In the former case, the first program version of the task is assigned to the returned group for execution. The above process is repeated until the task is aborted, the task is aborted and added to the aborted task queue Q_a for later execution, or the output for the task is obtained. The formal algorithm is given in [12].

1.3.4. Integrated Fault-Tolerant (FCFS + First Fits First) Scheduling Algorithm

The Integrated Fault-Tolerant (FCFS + first Fits First) (FCFSFFF) scheduling algorithm works as follows. As tasks, which may consist of more than one program version, arrive at the system, they are queued up along with their group sizes (i.e., g_i = task hardware reliability degree t_{hi} + 1) and task software reliability degree t_{si} in a single task queue Q. When a task T_i is scheduled for execution, the Dynamic Group Maximum Matching (DGMM) algorithm is called to find the required group size for the task T_i. If the returned group size by the DGMM algorithm is smaller than the required group size, then the returned group is allocated to the first program version V_{1j} of the first task T_j in the task queue that fits the returned group. Next, the DGMM algorithm is called to find another subgraph of size g_i in a different part of the system graph to allocate the task T_i. This process is repeated until either a group of size g_i is obtained or the entire system graph is searched without success. In the latter case, the task T_i is added to the aborted task queue Q_a for later execution. In the former case, the first program version V_{1i} of the task T_i is assigned to the returned group for execution. When a program version V_{ji} of a task T_i completes its execution by all the processors of its group G_i, neighboring processors exchange and compare their outputs. Then, the disagreement graph DG_i is obtained. A task T_i is released if at least (t_{hi} + 1) different processors agree with each other on the output for at least (t_{si} + 1) different program versions and the outputs of all the program versions are the same; otherwise, if there are at least (t_{hi} + 1) processors agree with each other on the output for ($2t_{si}$) or fewer different program versions and there are two or more different outputs, then the next program version of the task T_i is assigned to the group G_i for execution. Otherwise, if there are at least (t_{hi} + 1) processors which agree with each other on the outputs of ($2t_{si}$ + 1) different program versions and there are three or more different outputs, then the task T_i is aborted; otherwise, the task group size is incremented by one ($g_i = g_i + 1$), and the DGMM

algorithm is called to add one more neighboring processor to the group G_i. Calling the DGMM algorithm is repeated until either a group of the required size is obtained or the entire graph is searched without success. In the latter case, the task T_i is aborted and added to the aborted task queue Q_a for later execution. In the former case, the first program version of the task is assigned to the returned group for execution. The above process is repeated until the task is aborted, the task is aborted and added to the aborted task queue Q_a for later execution, or the output for the task is obtained. The formal algorithm is given in [12].

2. Simulation Model

The features of the simulator are summarized as follows [12]:

1. The computing environment is an $M \times M$ *torus* system ($M \geq 1$) connected to a host machine where scheduling and obtaining tasks disagreement graphs take place.
2. Each task (program) T_i which arrives at the system along with its reliability degree t_i will be assigned to a group G_i of size g_i (initially $g_i = t_i + 1$).
3. Tasks interarrival times are exponentially distributed with the average arrival rate λ.
4. Tasks mean execution times are exponentially distributed. Tasks arrived at the system could have different mean execution times.

3. Simulation Results

In our simulation we consider a 6 x 6 torus system *(M* = 6). See Figure 1. We assume that there are long tasks *and* short tasks. Mean execution time of long task is 10 *time units* and mean execution time of short task is 1 *time unit*. Tasks arrive at the system with the probability of being of long task equal to (*X*) and being of short task equal to (1-*X*); in other words, task length probability has a *Bernoulli* probability distribution. All the results given in this section assume $X = 0.5$. Also, we assume that the task software reliability $t_{si} = 1$ (each task has three program versions, with at least two fault free versions). The probability that the first two versions of a task being fault free (third version will not be executed) equal to (*Y*); in other words, third version execution probability has a *Bernoulli* probability distribution. All the results given in this section assume $Y = 0.5$. Furthermore, we assume that there are three types of task hardware reliability degrees: $t_{hi} = 0$ ($type_0$), $t_{hi} = 1$ ($type_1$) and $t_{hi} = 2$ ($type_2$). Tasks arrive at the system with the probability of being of $type_0$ equal to (Z_0), of being of $type_1$ equal to (Z_1), and of being of $type_2$ equal to (Z_2). In other words, tasks hardware reliability degrees probability has a *Binomial* probability distribution. All the results given in this section assume $Z_i = 1/3$, for $i = 0, 1, 2$. Each processor in the system has the probability (reliability) of being fault free equal to (*Rp*); in other words, processor reliability has a *Bernoulli* probability distribution. Each communication link in the system has the probability of being fault-free equal to (R_l); in other words, communication link reliability has also a *Bernoulli* probability distribution. In our simulation, we consider four failure cases with each type of tasks software reliability. First case, processors and communication links are fault-free, $Rp = 1$ and $R_l = 1$. Second case, only communication links are subject to failures, $Rp = 1$ and $R_l = 0.9$. Third case, only processors are subject to failures, $Rp = 0.9$ and $R_l = 1$. Fourth case, both processors and communication links are subject to failures, $Rp = 0.9$ and $R_l = 0.9$.

Our simulation terminates when the number of tasks released by the system is equal to 3000 tasks. The first 300 tasks released by the system are discarded, so the initial transient state of the system does not affect the simulation results. Each performance metric reading is an average over 10 runs.

We evaluate two performance metrics. The first metric is system mean response time. The second metric is percentage of tasks of $type_i$ completed, for $i = 0, 1, 2$. This metric is defined as follows:

percentage of tasks of $type_i$ *completed during simulation time* =

$$\frac{number\ of\ tasks\ of\ type_i\ completed}{number\ of\ tasks\ of\ type_i\ arrived} \times 100 \quad (1)$$

This metric is intended to complement the former performance metric. For instance, if a scheduling policy favors running the shorter tasks over the longer tasks for improving the system mean response time, one is interested to know what the trade offs. In other words; how much the percentage of longer tasks completed during the simulation time is decreased.

3.1. FCFS Scheduling Algorithm Performance

Figure 2 shows system average response time under the Integrated Fault-Tolerant First-Come, First-Served (FCFS) scheduling algorithm. From the plot we can see that as the task arrival rate λ increases, the average response time also increases.

Figures 3, 4, 5 and 6 show the percentage of tasks of $type_i$ completed, for $i = 0, 1, 2$, by FCFS scheduling algorithm, under each one of the four failure cases respectively. From the plots we can see that when the task arrival rate λ equal to 1, the percentage of tasks completed of each tasks type under the four failure cases is almost the same. Also, we can see that as arrival rate λ increases, the percentage of tasks completed of each tasks type decreases. Furthermore, from the figures we can see that the percentage of tasks completed of all tasks types under each one of the failure cases is almost the same. In other words, FCFS does not favor one type of task over another type of task for execution.

3.2. FCFSSFF Scheduling Algorithm Performance

Figure 7 shows system average response time under the Integrated Fault-Tolerant (FCFS + Smallest Fits First) (FCFSSFF) scheduling algorithm. In Figure 7, up

to a point (in our experiment arrival rate = 2) as task arrival rate λ increases, the system average response time also increases. Beyond that point, as arrival rate increases, the system average response time decreases. This is due to the fact that when the task arrival rate λ is high, more tasks will be queued up in the task queue and if the returned group size by the DGMM algorithm is smaller than the required size, the FCFSSFF scheduling algorithm will assign the returned group to the task with the smallest group size in the task queue. This means that tasks with small group sizes will be executed first; i.e., more concurrent tasks running on the system. With a higher task arrival rate (in our experiment arrival rate > 5), as arrival rate increases, the system average response time also increases. This is due to that the fact the length of the task queue will grow longer. Thus, even tasks with small group size have to wait longer in the task queue before being schedule for execution.

Figures 8, 9, 10 and 11 show the percentage of tasks of $type_i$ completed, for i = 0, 1, 2, by FCFSSFF scheduling algorithm, under each one of the four failure cases respectively. From the plots we can see that when the task arrival rate λ equal to 1, the percentage of tasks completed of each tasks type under the four failure cases is almost the same. Also, we can see that as arrival rate λ increases, the percentage of tasks completed of each tasks type decreases. Furthermore, we can see that as the task arrival rate λ increases, the percentage of tasks completed with large group is lower than the percentage of tasks completed with small group. This is due to the fact that when the task arrival rate λ is high, more tasks will be queued up in the task queue and if the returned group size by the DGMM algorithm is smaller than the required size, the FCFSSFF scheduling algorithm will assign the returned group to the task with the smallest group size in the task queue. This means that tasks with small group sizes will be executed first, in other words, FCFSSFF scheduling algorithm favors tasks with small group over tasks with large group for execution.

3.3. FCFSLFF Scheduling Algorithm Performance

Figure 12 shows system average response time under the Integrated Fault-Tolerant (FCFS + Largest Fits First) (FCFSLFF) scheduling algorithm. In Figure 12, we can see that as task arrival rate increases the system average response time also increases.

Figures 13, 14, 15 and 16 show the percentage of tasks of $type_i$ completed, for i = 0, 1, 2, by FCFSLFF scheduling algorithm, under each one of the four failure cases respectively. From the plots we can see that when the task arrival rate λ equal to 1, the percentage of tasks completed of each tasks type under the four failure cases is almost the same. Also, we can see that as arrival rate λ increases, the percentage of tasks completed of each tasks type decreases. Furthermore, from the Figure 13 we can see that the percentage of tasks completed of all tasks types under the first failure case is almost the same. Contrarily in Figures 14, 15 and 16 we can see that as the task arrival rate λ increases, the percentage of tasks completed with large group is lower than the percentage of tasks completed with small group. This is due to the fact that when the task arrival rate λ is high, more tasks will be queued up in the task queue and under the last three failure cases, the DGMM algorithm will return small group sizes, thus, the FCFSLFF scheduling algorithm will assign the returned group to task with small group size. This means that if the system contains faulty components, FCFSLFF scheduling algorithm favors tasks with small group over tasks with large group for execution.

3.4. FCFSFFF Scheduling Algorithm Performance

Figure 17 shows system average response time under the Integrated Fault-Tolerant (FCFS + first Fits First) (FCFSFFF) scheduling algorithm. In Figure 17, as task arrival rate increases the system average response time also increases.

Figures 18, 19, 20 and 21 show the percentage of tasks of $type_i$ completed, for i = 0, 1, 2, by FCFSFFF scheduling algorithm, under each one of the four failure cases respectively. From the plots we can see that when the task arrival rate λ equal to 1, the percentage of tasks completed of each tasks type under the four failure cases is almost the same. Also, we can see that as arrival rate λ increases, the percentage of tasks completed of each tasks type decreases. Furthermore, from the Figure 18 we can see that the percentage of tasks completed of all tasks types under the first failure case is almost the same. Contrarily in Figures 19, 20 and 21 we can see that as the task arrival rate λ increases, the percentage of tasks completed with large group is lower than the percentage of tasks completed with small group. This is due to the fact that when the task arrival rate λ is high, more tasks will be queued up in the task queue and under the last three failure cases, the DGMM algorithm will return small group sizes, thus, the FCFSFFF scheduling algorithm will assign the returned group to task with small group size. This means that if the system contains faulty components, FCFSFFF scheduling algorithm favors tasks with small group over tasks with large group for execution.

4. Conclusion

In this work, via four scheduling algorithms, the performance of the Integrated Fault-Tolerant (IFT) technique was studied. Two performance metrics were evaluated: system average response time and percentage of completed tasks of specific type.

Under the Integrated Fault-Tolerant First-Come, First-Served (FCFS) scheduling algorithm, our simulation study showed that under the conditions experimented here, as arrival rate λ increases, the system average response time also increases.

Under the Integrated Fault-Tolerant First-Come, First-Served + Smallest Fits First (FCFSSFF) scheduling algorithm, our simulation study showed that under the conditions experimented here, beyond a point, as arrival rate λ increases, the system average response time decreases. With a higher task arrival rate, the system average response time increases.

Under the Integrated Fault-Tolerant First-Come, First-Served + Largest Fits First (FCFSLFF) scheduling

algorithm, our simulation study showed that under the conditions experimented here, as arrival rate λ increases, the system average response time also increases.

Under the Integrated Fault-Tolerant First-Come, First-Served + First Fit First (FCFSFFF) scheduling algorithm, our simulation study showed that under the conditions experimented here, as arrival rate λ increases, the system average response time also increases.

Also, the study showed that FCFSSFF scheduling algorithm average response time outperforms the other algorithms. Furthermore, it showed that FCFS scheduling algorithm gives the highest average response time.

In addition, the study showed that FCFSSFF scheduling algorithm, FCFSLFF scheduling algorithm and FCFSFFF scheduling algorithm favor tasks with small group sizes over tasks with large group sizes for execution. Contrarily, FCFS scheduling algorithm does not favor tasks with small group over tasks with large group sizes for execution.

5. References

[1] O. Serlin. Fault-Tolerant System in Commercial Applications. IEEE Computer, Volume 17: 19-30, 1984.

[2] D. A. Rennels. Fault-Tolerant Computing-Concepts and Examples. IEEE Transactions on Computers, Volume C-33: 1116-1129, 1984.

[3] D. P. Siewiorek. Architecture of Fault-Tolerant Computers. IEEE Computer, Volume 17: 9-18, 1984.

[4] P. Jalote. Fault Tolerance in Distributed Systems. PTR Prentice Hall, 1994.

[5] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, B. Randell. A Program Structure for Error Detection and Recovery. Lecture Notes in Computer Science, Volume 16: 171-187, Springer-Verlag, New York, 1974.

[6] B. Randell. System Structure for Software Fault Tolerance. IEEE Transactions on Software Engineering, Volume SE-1: 220-232, 1975.

[7] T. Anderson, D. N. Halliwell, P. A. Barrett, M. R. Moulding. An Evaluation of Software Fault Tolerance in a Practical System. In proceedings of the 15th International Symposium on Fault Tolerant Computing, 1985.

[8] R. H. Campbell, K. H. Horton, G. G. Belford. Simulations of a Fault-Tolerant Deadline Mechanism. In proceedings of the *9th* International Symposium on Fault Tolerant Computing, 1979.

[9] H. Hecht. Fault-Tolerant Software. IEEE Transactions on Reliability, Volume R-28: 227-232, 1979.

[10] H. O. Welch. Distributed Recovery Block Performance in a Real-Time Control Loop. In proceedings of the Real-Time Systems Symposium, 1983.

[11] L. L. Pullum. Software Fault Tolerance Techniques and Implementation. Artech House, 2001.

[12] O. A. Abulnaja, High Performance Techniques for Reliable Execution of Tasks Under Hardware and Software Faults. Ph.D. Thesis, University of Wisconsin-Milwaukee, 1996.

[13] S. H. Hosseini. Fault-Tolerant Scheduling of Independent Tasks and Concurrent Fault-Diagnosis in Multiple Processor Systems. In proceedings of the IEEE International Conference on Parallel Processing, Volume I, 1988.

[14] S. H. Hosseini and T. P. Patel, An Efficient and Simple Algorithm for Group Maximum Matching. In proceedings of the 4th ISMM/IASTED International Conference on Parallel and Distributed Computing Systems, 1991.

Biography:

O. A. Abulnaja received the BS degree in computer science from King Abdulaziz University, Jeddah, Saudi Arabia, in 1986. He received the MS degree in computer science and the PhD degree in engineering (computer science) from University of Wisconsin-Milwaukee, Milwaukee, Wisconsin, USA, in 1990 and 1996, respectively. Currently, he is an associate professor of computer science at King Abdulaziz University. His research interests are Fault Tolerance, Systems Programming, Software Engineering, Parallel and Distributed Processing, and Systems Performance.

N. M. Saadi received the BS degree in computer science from King Abdulaziz University, Jeddah, Saudi Arabia, in 2001. He works for Faculty of Technology. His research interests are Fault Tolerance, Systems Programming, and Systems Performance.

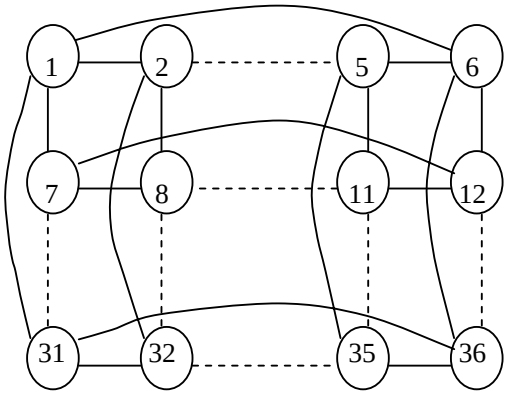

Figure 1: 6 × 6 Torus System

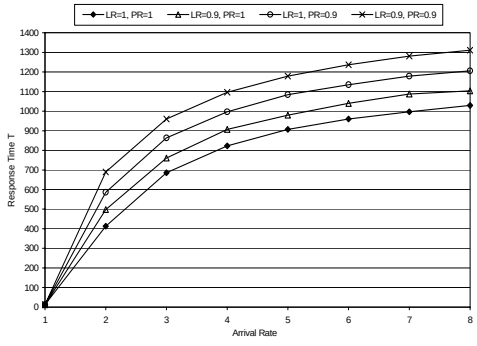

Figure 2: System Mean Response Time under FCFS, t_{ti} =0,1,2

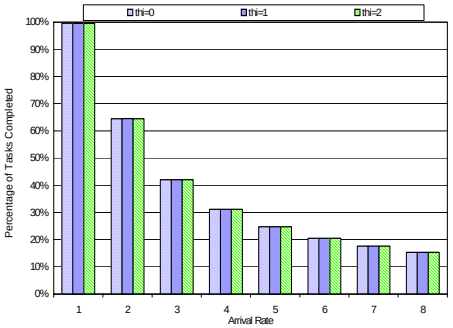

Figure 3: Percentage of Task completed under FCFS, LR=1, PR=1

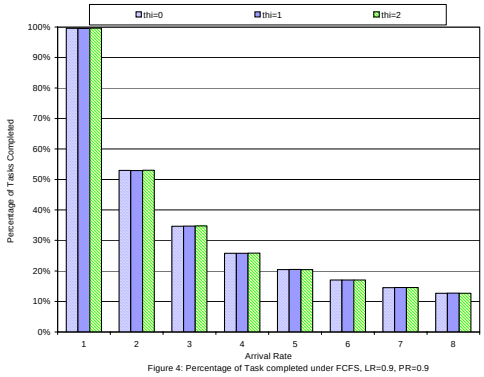

Figure 4: Percentage of Task completed under FCFS, LR=0.9, PR=0.9

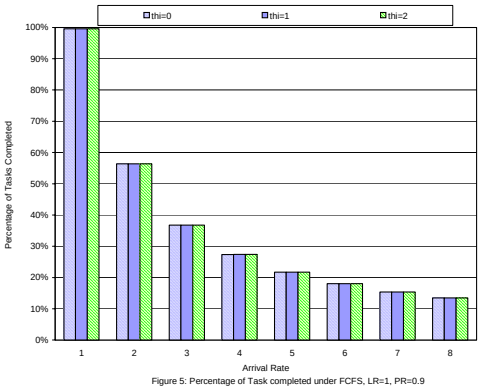

Figure 5: Percentage of Task completed under FCFS, LR=1, PR=0.9

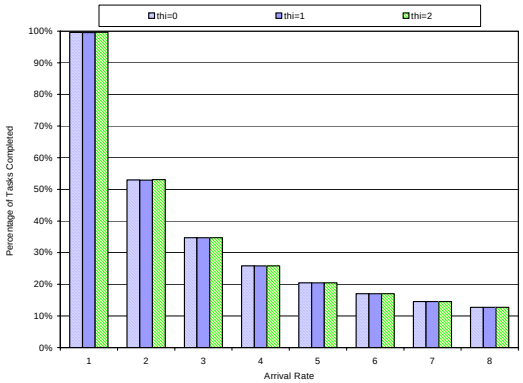

Figure 6: Percentage of Task completed under FCFS, LR=0.9 , PR=0.9

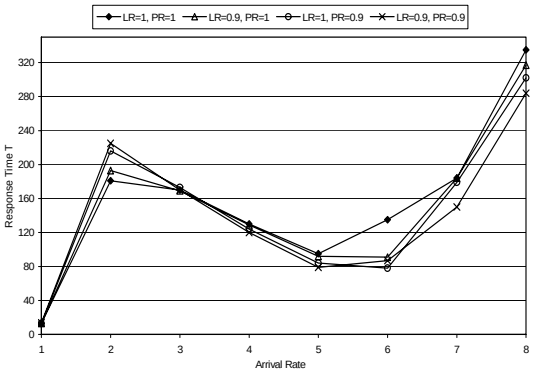

Figure 7: System Mean Response Time under FCFSSFF, t_{ti} =0,1,2

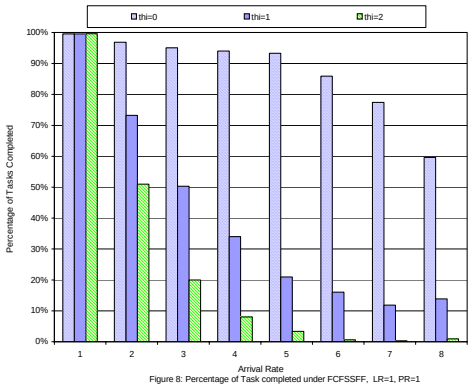

Figure 8: Percentage of Task completed under FCFSSFF, LR=1, PR=1

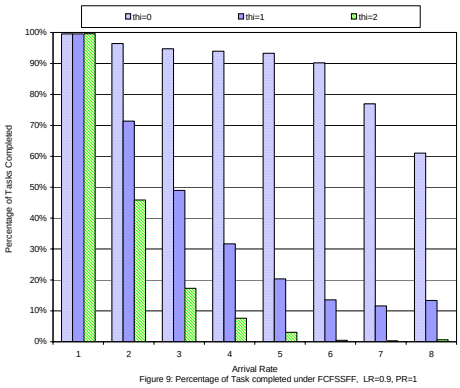

Figure 9: Percentage of Task completed under FCFSSFF, LR=0.9, PR=1

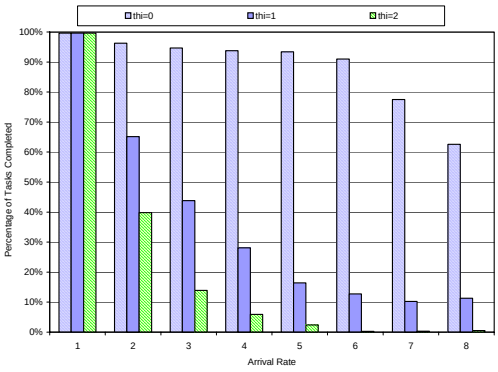

Figure 10: Percentage of Task completed under FCFSSFF, LR=1, PR=0.9

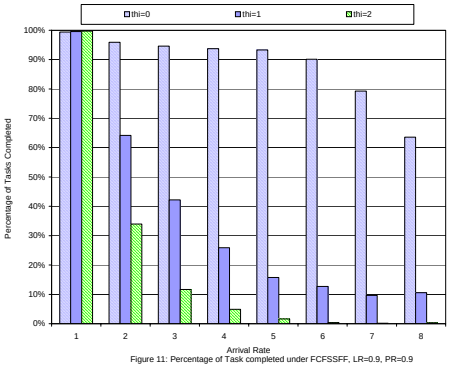

Figure 11: Percentage of Task completed under FCFSSFF, LR=0.9, PR=0.9

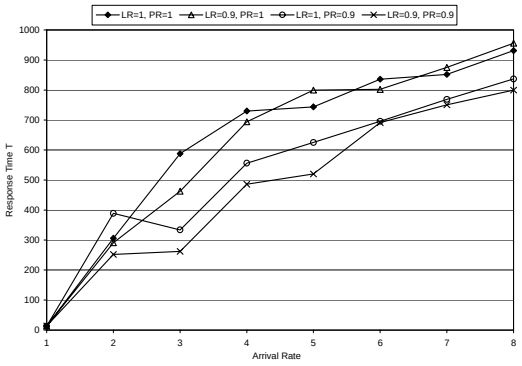

Figure 12: System Mean Response Time Under FCFSLFF, t_{tj} = 0, 1, 2

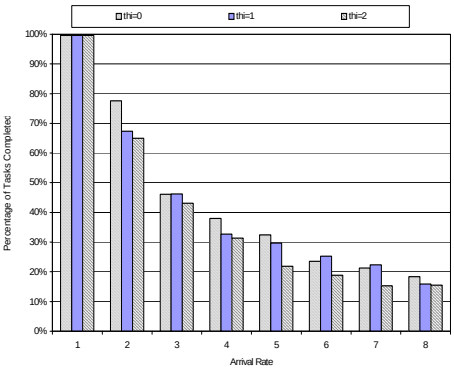

Figure 13: Percentage of Task completed under FCFSLFF, LR=1 , PR=1

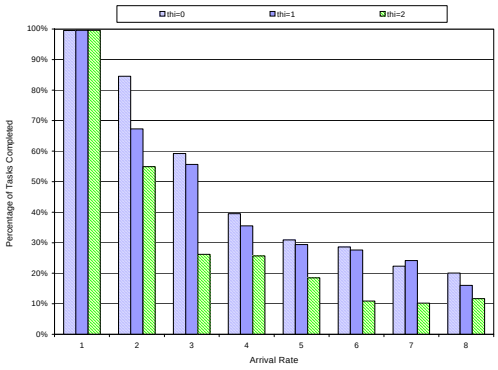

Figure 14: Percentage of Task completed under FCFSLFF, LR=0.9 , PR=1

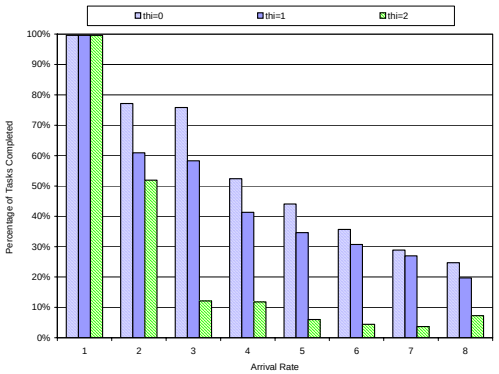

Figure 15: Percentage of Task completed under FCFSLFF, LR=1, PR=0.9

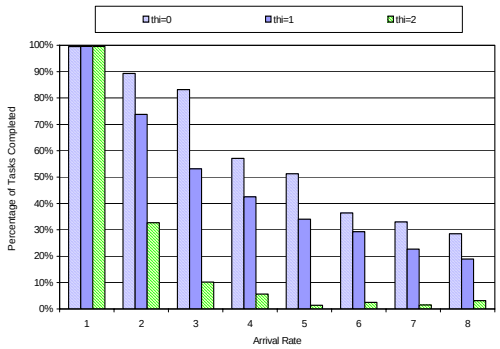

Figure 16: Percentage of Task completed under FCFSLFF,LR=0.9 , PR=0.9

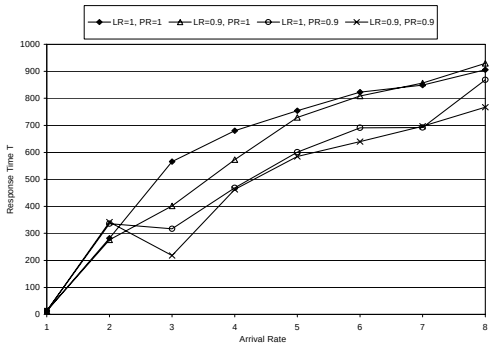

Figure 17: System Mean Response Time Under FCFSFFF, t_{fo} = 0, 1, 2

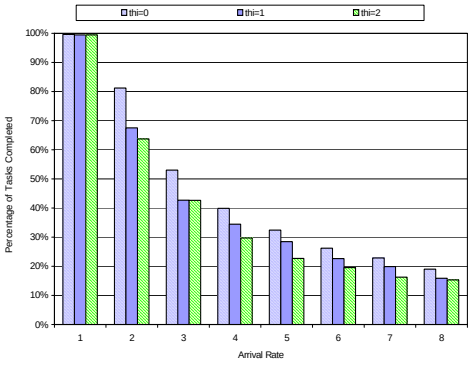

Figure 18: Percentage ofTask completed under FCFSFFF, LR=1 , PR=1

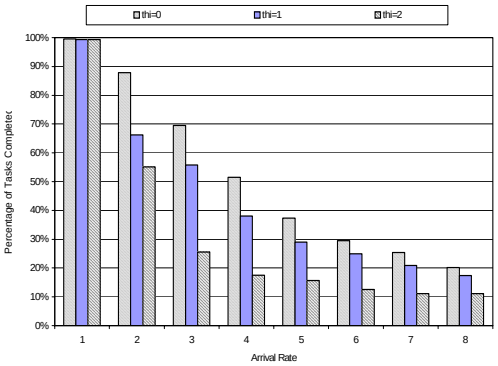

Figure 19: Percentage of Task completed under FCFSFFF, LR=0.9 , PR=1

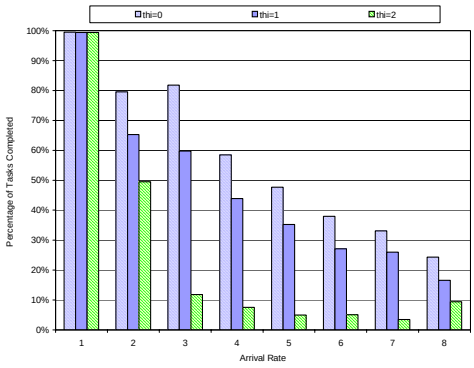

Figure 20: Percentage ofTask completed under FCFSFFF, LR=1 , PR=0.9

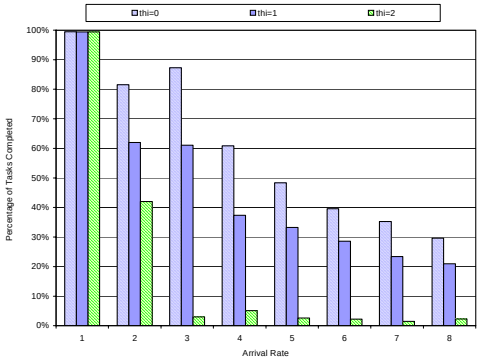

Figure 21: Percentage ofTask completed under FCFSFFF, LR=0.9 , PR=0.9