Fault Tolerant Scheduling of Precedence Task Graphs on Heterogeneous Platforms

Anne Benoit, Mourad Hakem, Yves Robert

▶ To cite this version:

Anne Benoit, Mourad Hakem, Yves Robert. Fault Tolerant Scheduling of Precedence Task Graphs on Heterogeneous Platforms. [Research Report] RR-6418, 2008. inria-00207593v2

HAL Id: inria-00207593
https://inria.hal.science/inria-00207593v2

Submitted on 21 Jan 2008 (v2), last revised 22 Jan 2008 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Fault Tolerant Scheduling of Precedence Task Graphs on Heterogeneous Platforms

Anne Benoit — Mourad Hakem — Yves Robert

N° ????

December 2007

Thème NUM

ISSN 0249-6399   ISRN INRIA/RR--????--FR+ENG

Fault Tolerant Scheduling of Precedence Task Graphs on Heterogeneous Platforms

Anne Benoit, Mourad Hakem, Yves Robert

Thème NUM — Systèmes numériques
Projet GRAAL

Rapport de recherche n° ???? — December 2007 — 24 pages

Abstract: Fault tolerance and latency are important requirements in several applications which are time critical in nature: such applications require guaranties in terms of latency, even when processors are subject to failures. In this paper, we propose a fault tolerant scheduling heuristic for mapping precedence task graphs on heterogeneous systems. Our approach is based on an active replication scheme, capable of supporting ε arbitrary fail-silent (fail-stop) processor failures, hence valid results will be provided even if ε processors fail. We focus on a bi-criteria approach, where we aim at minimizing the latency given a fixed number of failures supported in the system, or the other way round. Major achievements include a low complexity, and a drastic reduction of the number of additional communications induced by the replication mechanism. Experimental results demonstrate that our heuristics, despite their lower complexity, outperform their direct competitor, the FTBAR scheduling algorithm [8].

Key-words: Fault tolerance, reliability, multi-criteria scheduling, heterogeneous systems.

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme http://www.ens-lyon.fr/LIP.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Ordonnancement tolérant aux pannes de graphes de tâches sur plates-formes hétérogènes

Résumé : La tolérance aux pannes et la latence sont deux critères importants pour plusieurs applications qui sont critiques par nature. Ce type d'applications exige des garanties en terme de temps de latence, même lorsque les processeurs sont sujets aux pannes. Dans ce rapport, nous proposons une heuristique tolérante aux pannes pour l'ordonnancement de graphes de tâches sur des systèmes hétérogènes. Notre approche est basée sur un mécanisme de réplication active, capable de supporter ε pannes arbitraires de type silence sur défaillance. En d'autres termes, des résultats valides seront fournis même si ε processeurs tombent en panne. Nous nous concentrons sur une approche bi-critère, où nous avons pour objectif de minimiser le temps de latence pour un nombre donné (fixé) de pannes tolérées dans le système, ou l'inverse. Les principales contributions incluent une faible complexité en temps d'exécution, et une réduction importante du nombre de communications induites par le mécanisme de réplication. Les résultats expérimentaux montrent que notre algorithme, en dépit de sa faible complexité temporelle, est meilleur que son direct compétiteur, l'algorithme FTBAR [8].

Mots-clés : Tolérance aux pannes, fiabilité, ordonnancement multicritère, ressources hétérogènes.

1 Introduction

Heterogeneous distributed systems are widely deployed for executing computationally intensive parallel applications with diverse computing needs. The efficient execution of applications in such environments requires effective scheduling strategies that take into account both algorithmic and architectural characteristics. The goal is to achieve a good mapping of tasks to processors, minimizing the schedule length (latency). In addition, resource failures (processors/links) may frequently occur in such systems and have an adverse effect on applications. Consequently, there is an increasing need for developing techniques to achieve fault tolerance, *i.e.*, to tolerate an arbitrary number of failures during the execution. Both heterogeneous scheduling and fault tolerance are difficult problems in their own, and aiming at solving them together makes the problem even harder. For instance, the latency of the application will increase if we want to tolerate several failures, even if no actual failure happens during execution.

In this paper, we introduce a Fault Tolerant Scheduling Algorithm (FTSA) which aims at tolerating multiple processor failures without sacrificing the latency simultaneously. FTSA is based on an active replication scheme to mask failures, so that there is no need for detecting and handling such failures. Major achievements include a low complexity, and a drastic reduction of the number of additional communications induced by the replication mechanism in the MC-FTSA variant of the algorithm (where MC stands for *Minimum Communications*). Experimental results demonstrate that our heuristics, despite their lower complexity, outperform their direct competitor, the FTBAR scheduling algorithm [8].

Throughout the paper, we will use terms latency, makespan and schedule length indifferently.

The paper is organized as follows: Section 2 presents basic definitions and assumptions. We overview related work in Section 3. Section 4 describes FTSA, together with its variant MC-FTSA designed to minimize communication overhead. We outline the principle of FTBAR [8] in Section 5, and we compare FTSA to the latter algorithm in Section 6; the experimental results assess the good behavior of our algorithms. Finally, we conclude in Section 7.

2 Framework

The execution model for a task graph is represented as a weighted Directed Acyclic Graph (DAG) $G = (V, E)$, where V is the set of nodes corresponding to the tasks,

and E is the set of edges corresponding to the precedence relations between the tasks. In the following we use the term node or task indifferently; $v = |V|$ is the number of nodes, and $e = |E|$ is the number of edges. In a DAG, a node without any predecessor is called an *entry* node, while a node without any successor is an *exit* node. For a task t in G, $\Gamma^-(t)$ is the set of immediate predecessors and $\Gamma^+(t)$ denotes its immediate successors. We let $\mathcal{V}$ be the edge cost function: $\mathcal{V}(t_i, t_j)$ represents the volume of data that task t_i needs to send to task t_j.

A heterogeneous system is represented by a finite processor set $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_m\}$. These processors are assumed to be fully connected. The link between processors $\mathcal{P}_k$ and $\mathcal{P}_h$ is denoted by ℓ_{kh}. The computational heterogeneity of tasks is modeled by a function $\mathcal{E} : V \times P \rightarrow R^+$, which represents the execution time of each task on each processor in the system: $\mathcal{E}(t, \mathcal{P}_k)$ denotes the execution time of t on $\mathcal{P}_k$, $1 \leq k \leq m$. The heterogeneity in terms of communications is expressed by $W(t_i, t_j) = \mathcal{V}(t_i, t_j).d(\mathcal{P}_k, \mathcal{P}_h)$, where task t_i is mapped on processor $\mathcal{P}_k$, task t_j is mapped on processor $\mathcal{P}_h$, and $d(\mathcal{P}_k, \mathcal{P}_h)$ is the time required to send a unit length data from $\mathcal{P}_k$ to $\mathcal{P}_h$. The communication has no cost if the two tasks are mapped on the same processor: $d(\mathcal{P}_k, \mathcal{P}_k) = 0$.

The mapping matrix $\mathcal{X}$ is a $v \times m$ binary matrix representing the mapping of the v tasks in the DAG to the m processors. Element $\mathcal{X}_{ik}$ is 1 if task t_i has been mapped to processor $\mathcal{P}_k$ and 0 otherwise.

For a given graph G and processor set $\mathcal{P}$, $g(G, \mathcal{P})$ is the *granularity*, *i.e.*, the ratio of the sum of slowest computation times of each task, to the sum of slowest communication times along each edge. If $g(G, \mathcal{P}) \geq 1$, the task graph is said to be *coarse grain*, otherwise it is *fine grain*. For *coarse grain* DAGs, each task receives or sends a small amount of communication compared to the computation of its adjacent tasks. During the scheduling process, the graph consists of two parts, the already examined (scheduled) tasks S and the unscheduled tasks U. Initially $U = V$.

Our goal is to find a task mapping $\mathcal{X}$ and a schedule of the DAG G on platform $\mathcal{P}$, which aims at minimizing the latency $\mathcal{L}(G, \mathcal{X})$, while tolerating an arbitrary number ε of processor failures.

3 Related Work

A large number of algorithms for scheduling and partitioning DAGs have been proposed in the literature, either with an unbounded number of processors [6, 9, 18, 23] or with a limited number of processors [3, 15, 24, 27]. All above references assume that processors in the systems are completely safe.

Reliability has been considered in [16, 17, 26]. Task allocation models which aim at maximizing the reliability of the system have been developed for heterogeneous systems. However, these heuristics neither provide fault tolerance nor attempt to minimize the latency of the application. Some other papers [4, 5, 10, 14, 21] deal with both objectives, performance (latency) and reliability. These algorithms are developed only for maximizing reliability while satisfying latency constraints. They do not achieve fault tolerance. Recall that, the reliability is a probability measure that evaluates by a probabilistic calculation the good behavior of a system. It is used juste to guarantee a minimum service of proper functioning. But the fault tolerance allows a system to continue to deliver a service even in the presence of failures.

In multiprocessor systems, fault tolerance can be provided by scheduling multiples copies (replicas) of tasks on different processors. A large number of techniques for supporting fault-tolerant systems have been proposed [2, 7, 8, 11, 12, 19, 20, 22, 28]. There are two main approaches, as described below.

(i) Primary/Backup (passive replication).
This is the traditional fault-tolerant approach where both time and space exclusions are used. The main idea of this technique is that the backup task is activated only if the fault occurs while executing the primary task [20, 28]. This technique also assumes that there is a fault detection mechanism that detects a processor crash. The main disadvantage of this scheme is that only two copies of the task are scheduled on different processors (space exclusion) with time exclusion. To achieve high schedulability while providing fault-tolerance, the heuristics presented in [2, 7, 19] apply two techniques while scheduling the primary and backup copies of the tasks:
- *backup overloading*: scheduling backups for multiple primary tasks during the same time slot in order to make efficient utilization of available processor time, and
- *de-allocation* of resources reserved for backup tasks when the corresponding primaries complete successfully.
Note that this technique can be applied only under the assumption that only one processor may fail at a time. The overloading technique is quite simple in this context, because if two backups bt_i and bt_j of tasks t_i and t_j respectively, are scheduled on the same processor, then these backups can overlap since $\text{proc}(t_i)$ and $\text{proc}(t_j)$ will not fail at the same time. All algorithms belonging to this categorie [2, 7, 19, 20, 28] share three common points: (i) tasks have deadlines and are independent, (ii) the system architecture is homogeneous, and (iii) they support only one processor failure.

Recently, Xiao and Hong proposed a scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems [22]. Once more, the algorithm

is devised to handle only one processor failure. The tasks are assumed to be non-preemptable, and each task has two copies that are scheduled on different processors and mutually excluded in time. The quality of the schedule is achieved by allowing a backup copy to overlap with other backup copies on the same processor since they consider at most one processor failure. This algorithm takes a reliability measure of the system into account . But the two objectives (reliability and performance) are not considered simultaneously. First, the algorithm tries to guarantee the timing constraints (deadlines) of the tasks. Then, among the processors on which the deadline of a task is guaranteed, the task is mapped to the processor which minimizes the failure probability of the application. However, deadlines have priority over the reliability objective.

To summarize, all these techniques assume that only one processor can fail at any time and that a second processor cannot fail before the system recovers from the first failure.

(ii) Active replication (N-Modular redundancy).
This technique is based on space redundancy, *i.e.*, multiple copies of each task are mapped on different processors, which are run in parallel to tolerate a fixed number of failures. For instance, Hashimoto et al. [11, 12] propose an algorithm that tolerates one processor failure on homogeneous system. This algorithm exploits implicit redundancy (originally introduced by task duplication in order to minimize the schedule length) and assumes that some processors are reserved only for realizing fault tolerance, *i.e.*, the reserved processors are not used for the original scheduling. Girault et al. present FTBAR, a static real-time and fault-tolerant scheduling algorithm where multiple processor failures are considered [8]. To the best of our knowledge, FTBAR is the closest work to the one presented in this paper. A brief description of FTBAR is given in Section 5, and we experimentally compare it to FTSA in Section 6.

4 FTSA and MC-FTSA

In this section, we present FTSA (Fault Tolerant Scheduling Algorithm), whose objective is to minimize the latency $\mathcal{L}(G, \mathcal{X})$ while tolerating an arbitrary number ε of fail-silent (fail-stop) processor failures under task mapping $\mathcal{X}$. FTSA uses an active replication strategy to allocate $\varepsilon+1$ copies of each task to different processors.

Allocating many copies of each task will severely increase the total number of communications required by the algorithm: we move from e communications (one per edge) in a mapping with no replication, to $e(\varepsilon + 1)^2$ in FTSA, a quadratic

increase. We show how to reduce this overhead down to a linear number $e(\varepsilon + 1)$ of communications in the design of MC-FTSA (where MC stands for *Minimum Communications*).

4.1 FTSA

FTSA is a greedy scheduling heuristic based on an attribute priority called *task criticalness*, which is defined as the length of the longest path passing through free tasks in the current partially mapped DAG. Recall that a task is free if it is unscheduled and if all of its predecessors are scheduled. S is the set of scheduled tasks, U the set of unscheduled tasks, and $U_f \subseteq U$ the set of free tasks. Once a task $t \in S$ is scheduled on processor $\mathcal{P}(t)$, we know its start time $\mathcal{S}(t, \mathcal{P}(t))$ and its finish time $\mathcal{F}(t, \mathcal{P}(t))$.

A *critical task* is defined as one of the free tasks with the highest priority. The priority of a free task t is determined by $t\ell(t) + b\ell(t)$, where $t\ell(t)$ and $b\ell(t)$ are respectively the *dynamic top level* and the *static bottom level* of t. They are computed as follows:

$\forall\ t \in U_f$,
if $\Gamma^-(t) = \emptyset$ then $t\ell(t) \leftarrow 0$ else
$t\ell(t) \leftarrow \max\limits_{t^* \in \Gamma^-(t)} \left\{ \mathcal{F}(t^*, \mathcal{P}(t^*)) + \mathcal{V}(t^*, t). \max\limits_{1 \le j \le m} d(\mathcal{P}(t^*), \mathcal{P}_j) \right\}$

$\forall\ t \in U$,
if $\Gamma^+(t) = \emptyset$ then $b\ell(t) \leftarrow \overline{\mathcal{E}(t)}$
else $b\ell(t) \leftarrow \max\limits_{t^* \in \Gamma^+(t)} \{ \overline{\mathcal{E}(t)} + \overline{W(t, t^*)} + b\ell(t^*) \}$

The word *dynamic* implies that the value $t\ell$ depends upon the tasks which have already been mapped at each step in the mapping process and the word *static* implies that the value $b\ell$ remains unchanged according to the topological traversal (top-down) of the DAG.

In the computation of top levels, we consider the worst case communication since we do not know on which processor task t will be assigned. For bottom levels, we use the average execution time of t, defined as $\overline{\mathcal{E}(t)} = \left(\sum\limits_{j=1}^{m} \mathcal{E}(t, \mathcal{P}_j) \right) / m$, and the average communication cost of the edge (t, t_*), defined as $\overline{W(t, t^*)} = \mathcal{V}(t, t^*).\overline{d}$, where $\overline{d}$ is the average delay to send a unit length data between two processors in the system.

Note that the finish time of t on $\mathcal{P}_j$ is $\mathcal{F}(t, \mathcal{P}_j) = \mathcal{S}(t, \mathcal{P}_j) + \mathcal{E}(t, \mathcal{P}_j)$. The starting time $\mathcal{S}(t, \mathcal{P}_j)$ of t on $\mathcal{P}_j$ must be later than the time when all messages from t's predecessors arrive on processor $\mathcal{P}_j$, and also later than the ready time of processor $\mathcal{P}_j$, defined as $r(\mathcal{P}_j) = \max_{t_i \in S} \Big(\mathcal{X}_{ij} \mathcal{F}\big(t_i, \mathcal{P}_j\big)\Big)$. Thus, $\mathcal{S}(t, \mathcal{P}_j) = \max\big(t\ell^*(t, \mathcal{P}_j), r(\mathcal{P}_j)\big)$, where the top level is updated now that we know which processor task t is mapped onto:

$$t\ell^*(t, \mathcal{P}_j) = \max_{t^* \in \Gamma^-(t)} \big\{ \mathcal{F}(t^*, \mathcal{P}(t^*)) + W(t^*, t) \big\}$$

The definition of *criticalness* provides a good measure of the task importance: the greater the *criticalness*, the more work is to be performed along the path containing that task. FTSA takes the computational heterogeneity of the system into account and is designed with the following objectives: (i) tolerate an arbitrary number of permanent failures under latency constraints; (ii) compute task priorities accurately in order that critical tasks will finish earlier; (iii) have a low running time compared to other algorithms in the literature.

We maintain a priority list α (that contains free tasks) which is implemented by using a balanced search tree data structure (AVL). At the beginning, α is empty. The head function $\mathcal{H}(\alpha)$ returns the first task in the sorted list α, which is the task with the highest priority (ties are broken randomly). The number of tasks that can be simultaneously free at each step in the scheduling process is bounded by the *width* ω of the task graph (the maximum number of tasks that are independent in G). This implies that $|\alpha| \leq \omega$.

At each step of the mapping process, FTSA selects a critical free task t $\big(t \leftarrow \mathcal{H}(\alpha)\big)$ and simulates its mapping on all processors using the following equation:

$$\forall\ 1 \leq j \leq m,$$
$$\mathcal{F}(t, \mathcal{P}_j) = \mathcal{E}(t, \mathcal{P}_j) + \max\left(\max_{t_* \in \Gamma^-(t)} \left\{ \min_{k=1}^{\varepsilon+1} \big\{ \mathcal{F}(t_*^k, \mathcal{P}(t_*^k)) + W(t_*^k, t) \big\} \right\}, r(\mathcal{P}_j) \right) \quad (1)$$

The predecessor tasks $t_* \in \Gamma^-(t)$ are already scheduled onto $\varepsilon + 1$ distinct processors, and we denote by t_*^k, $1 \leq k \leq \varepsilon + 1$, the replicas of task t_*. The first $\varepsilon + 1$ processors that allow the *minimum finish time* of t are kept. This set called $\mathcal{P}^{(\varepsilon+1)}$ (the superscript $\varepsilon + 1$ indicates the cardinality) is defined as the $\varepsilon + 1$ processors $\mathcal{P}_j$ which realize the lowest value of finish time $\mathcal{F}(t, \mathcal{P}_j)$.

Once the set $\mathcal{P}^{(\varepsilon+1)}$ is determined, the task t is scheduled on $\varepsilon + 1$ distinct processors (replicas) $\mathcal{P} \in \mathcal{P}^{(\varepsilon+1)}$. Let $\hat{t}$ be an exit task (a task which does not

have any successors in G). The latency of the schedule generated using the above equation, represents a lower bound, *i.e.*, this latency can be achieved if no processor permanently fails during the execution of the application. It is defined as follow:

$$\mathcal{M}^* = \max_{\hat{t}_i \in S} \left\{ \min_{1 \le k \le \varepsilon+1} \left\{ \mathcal{F}(\hat{t}_i^k, \mathcal{P}(\hat{t}_i^k) \right\} \right\} \tag{2}$$

To compute the upper bound of the latency $\mathcal{M}$, which is achieved in the presence ε permanent failures (see proposition 4.2), we use the following formula:

$$\forall\ 1 \le j \le m,$$

$$\mathcal{F}(t, \mathcal{P}_j) = \mathcal{E}(t, \mathcal{P}_j) + \max\left(\max_{t_* \in \Gamma^-(t)} \left\{ \max_{k=1}^{\varepsilon+1} \left\{ \mathcal{F}(t_*^k, \mathcal{P}(t_*^k)) + W(t_*^k, t) \right\} \right\}, r(\mathcal{P}_j) \right) \tag{3}$$

Thus,

$$\mathcal{M} = \max_{\hat{t}_i \in S} \left\{ \max_{1 \le k \le \varepsilon+1} \left\{ \mathcal{F}(\hat{t}_i^k, \mathcal{P}(\hat{t}_i^k) \right\} \right\} \tag{4}$$

Proposition 4.1 *For an active replication scheme, a task $t_i \in G$ is guaranteed to execute in the presence of ε permanent faults if and only if $\mathcal{P}(t_i^k) \neq \mathcal{P}(t_i^{k+1}), k = 1 \ldots \varepsilon$.*

Proof: If ε processors fails, then $\mathcal{P}(t_i^z), 1 \le z \le \varepsilon + 1$, cannot fail and therefore $\mathcal{P}(t_i^z)$ will execute successfully since there are $\varepsilon + 1$ copies of t_i mapped on $\varepsilon + 1$ different processors. However, if there is a processor $\mathcal{P}(t_i^u), 1 \le u \le \varepsilon + 1$, such that $\mathcal{P}(t_i^u) = \mathcal{P}(t_i^z) = \mathcal{P}^*$ and $\mathcal{P}^*$ fails, then neither t_i^u nor t_i^z can execute successfully. □

Proposition 4.2 *The latency achieved by FTSA is $\mathcal{L} \le \mathcal{M}$ despite ε permanent failures.*

Proof: Each task $t \in G$ is replicated $\varepsilon + 1$ times. Each of these replicas send their data results to all replicas of each successors task. Therefore, each task will receive its input data $\varepsilon + 1$ times. But as soon as it receives the first input data, the task is executed and ignores later incoming data. So, in some cases the finish time of the replica $t^{(\varepsilon+1)}$ will be sooner than its estimated finish time computed by the formula, even in the presence of ε failures. Applying this reasoning to all tasks of G shows that $\mathcal{L} \le \mathcal{M}$. □

Theorem 4.1 *If at most ε processor failures occur in the system, then the schedule remains valid.*

Proof: FTSA is based on an active replication scheme with space exclusion. Thus, each task is replicated $\varepsilon + 1$ times onto $\varepsilon + 1$ distinct processors. We have at most ε processor failures at the same time. So at least one copy of each task is executed on a fault free processor. □

Note that if a replica of task t and a replica t_*^z of its predecessor t_* are mapped on the same processor $\mathcal{P}$, then there is no need for other copies of t_* to send data to processor $\mathcal{P}$. Indeed, if $\mathcal{P}$ is operational, then the copy of t on $\mathcal{P}$ will receive the data from t_*^z (intra-processor communication). Otherwise, $\mathcal{P}$ is down and does not need to receive anything.

Algorithm 4.1 The FTSA algorithm

1: $\varepsilon \leftarrow$ maximum number of failures supported in the system
2: Compute $b\ell(t)$ for each task t in G and set $t\ell(t) = 0$ for each entry task t;
3: $P = \{\mathcal{P}_1, \mathcal{P}_2, \ldots \mathcal{P}_m\}$; *(*Set of processors*)*
4: $S = \emptyset$; $U = V$; *(*Mark all tasks as unscheduled*)*
5: Put entry tasks in α;
6: **while** $U \neq \emptyset$ **do**
7: $t \leftarrow \mathcal{H}(\alpha)$; *(*Select free task with highest priority from α *)*
8: Compute $\mathcal{F}(t, \mathcal{P}_j)$ for $1 \leq j \leq m$ using equation (1);
9: Keep first $\varepsilon + 1$ processors that allow for minimum finish time of t, $\mathcal{P}^{(\varepsilon+1)}$;
10: Schedule t on these $\varepsilon + 1$ processors;
11: Put t in S and update priority values of t's successors;
12: Put t's free successors in α;
13: $U \leftarrow U \backslash \{t\}$;
14: **end while**

We are ready to assess the compexity of FTSA:

Theorem 4.2 *The time complexity of FTSA is:*

$$O(em^2 + v \log \omega)$$

Proof: Computing $b\ell(t)$ (line 2) takes $O(e + v)$. Insertion or deletion from α costs $O(\log |\alpha|)$ where $|\alpha| \leq \omega$. Since each task in a DAG is inserted into α once and only once and is removed once and only once during the entire execution of FTSA, the time complexity for α management is in $O(v \log \omega)$. The main computational cost of FTSA is spent in the while loop (Lines 6 to 14). This loop is executed v times. Line 7 costs $O(\log \omega)$ for finding the head of α. Line 8 costs $O(|\Gamma^-(t)|(\varepsilon + 1)m)$, since all

the replicas of the immediate predecessors of task t on each processor $\mathcal{P}_j, j = 1 \dots m$, need to be examined. Since $\varepsilon < m$, then for the whole v loops the cost for this line is at most $\sum_{i=1}^{v} O(|\Gamma^-(t)|m^2) = O(em^2)$. Line 11 costs $O(|\Gamma^+(t)|)$ to update the priority values of the immediate successors of t, and similarly, the cost for the v loops of this line is $O(e)$. Thus the total cost of FTSA is $O(em^2 + v \log \omega)$. □

4.2 MC-FTSA

Each task of the task graph G is replicated $\varepsilon+1$ times. Therefore each communication between two tasks in precedence is replicated at most $(\varepsilon + 1)^2$ times. Since there are e edges in G, the total number of messages in the fault tolerant schedule is at most $e(\varepsilon+1)^2$. In some cases, we may have an intra-processor communication, when two tasks in precedence are mapped on the same processor, so the latter quantity is in fact an upper bound. Duplicating each task $\varepsilon + 1$ times is an absolute requirement to resist to ε failures. But duplicating each precedence edge $(\varepsilon + 1)^2$ times is not mandatory. We can decrease the total number of communications from $e(\varepsilon + 1)^2$ down to $e(\varepsilon + 1)$, as explained below.

Let $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_m\}$ be the set of processors in the system. Let t be the current task scheduled by the algorithm FTSA. We use equation (1) to assign a set $\mathcal{A}(t)$ of $\varepsilon + 1$ processors to execute t. We need to orchestrate the communications from the processors executing the predecessors of t. So consider a predecessor t' of t, that has been scheduled on a set $\mathcal{A}(t')$ of $\varepsilon + 1$ processors. Now each processor in $\mathcal{A}(t')$ will communicate to exactly one processor in $\mathcal{A}(t)$ instead of communicating to all of them as in the FTSA algorithm. To determine the set of communications, we use a graph-theoretic approach. We prepare a bipartite graph as follows:
- left nodes are communication sources: we insert a vertex $v_{t',\mathcal{P}_i}$ for each $\mathcal{P}_i \in \mathcal{A}(t')$.
- right nodes are the target processors in $\mathcal{A}(t)$: we insert a vertex $v_{t,\mathcal{P}_i}$ for each $\mathcal{P}_i \in \mathcal{A}(t)$.
- edges go from left nodes to right nodes. Consider any left node $v_{t',\mathcal{P}_i}$. We have two cases: (i) either $\mathcal{P}_i \in \mathcal{A}(t)$, which means that there is a right node $v_{t,\mathcal{P}_i}$ in the graph, then we add an edge from $v_{t',\mathcal{P}_i}$ to $v_{t,\mathcal{P}_i}$, and this is the only edge outgoing from $v_{t',\mathcal{P}_i}$; (ii) or $P_i \notin \mathcal{A}(t)$, and we add an edge from $v_{t',\mathcal{P}_i}$ to each right node $v_{t,\mathcal{P}_j}$.
- an edge from v_{t',P_i} to v_{t,P_j} is weighted by the time-step at which the computation of t could be finished by $\mathcal{P}_j$ if t' was the only predecessor of t. More precisely, this weight is equal to

$$\max\left(\mathcal{F}(t', \mathcal{P}_i) + W(t', t), r(\mathcal{P}_j)\right) + \mathcal{E}(t, \mathcal{P}_j)$$

Recall that $W(t', t) = 0$ if $i = j$.

Proposition 4.3 *Any subset of* $\varepsilon+1$ *edges* $\mathcal{C}$ *such that each left node and each right node belongs to exactly one edge of* $\mathcal{C}$ *defines a robust set of communications,* i.e., *a set of communications capable to resist to* ε *processor failures.*

The intuitive idea is to enforce internal communications whenever a processor executes both t and one of its predecessor t', as we prove below. Note that the algorithm would fail otherwise. For instance assume that $\varepsilon = 2$, $\mathcal{A}(t') = \{\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \}$ and $\mathcal{A}(t) = \{\mathcal{P}_1, \mathcal{P}_5, \mathcal{P}_6\}$. If we retain the communications $\mathcal{P}_1 \rightarrow \mathcal{P}_5$, $\mathcal{P}_2 \rightarrow \mathcal{P}_6$ and $\mathcal{P}_3 \rightarrow \mathcal{P}_1$, then the algorithm is blocked by the failure of $\mathcal{P}_1$ and $\mathcal{P}_2$. But if we enforce that the only edge from $\mathcal{P}_1$ is to itself, then we resist to 2 failures.

Proof: Consider a predecessor t' of t, let $\mathcal{B} = \mathcal{A}(t') \cap \mathcal{A}(t)$ and $k = |\mathcal{B}|$. If one of the processors in $\mathcal{B}$ does not fail we are done. Otherwise there remains $\varepsilon + 1 - k$ processors in $\mathcal{A}(t')$, as many in $\mathcal{A}(t)$ and they are all distinct. We have $\varepsilon+1-k$ edges that realize a one-to-one mapping between these two processor sets. We can reorder the remaining processors in $\mathcal{A}(t)$ so that each edge go between the i-th remaining processor in $\mathcal{A}(t')$ to the i-th remaining processor in $\mathcal{A}(t)$. There can still be $\varepsilon - k$ failures, since we considered that all k processors in $\mathcal{B}$ have failed, and thus we cannot break the $\varepsilon - k + 1$ edges. There remains a communication link between two working processors. □

We need to decide which edge subset $\mathcal{C}$ to extract from the bipartite graph. As long as a subset satisfies to the condition of proposition 4.3, it is valid, but we aim at finding one that minimizes the latency. There are several possibilities:

- For any value of T, we can find in polynomial time if there exists a subset whose largest edge weight does not exceed T. To do so, we suppress all edges of weight larger than T, and we run a maximal matching algorithm (which is polynomial since the graph is bipartite) that will cover all source edges if such a cover exists, hence providing a valid solution. We perform a binary search on T to determine the smallest value that leads to a solution, which we return. Note that T is searched in the set of edge weights, hence the overall complexity of the algorithm remains polynomial.

- We can use a greedy algorithm that gives priority to internal communications and then greedily select the edges in the order of non-decreasing weights. We retain the current edge if it satisfies to the condition of proposition 4.3 given already taken decisions, *i.e.*, if it saturates a new left node and a new right node in the graph, and otherwise we proceed to the next edge.

4.3 With different objective functions

In this section we first discuss the approach when the latency is fixed (instead of the number of tolerated failures as before). When the latency is fixed, we would like to determine the maximum number of processor failures that can be tolerated in the system while achieving the prescribed latency. The simplest way is to start by generating a scheduling supporting a single failure, and if the length of the scheduling is less than the fixed latency, we repeat the process for 2 failures and so on until the latency requirement cannot be satisfied any longer. The running time of the scheduling process is increased since we perform several calls to FTSA. A better solution consists in performing a binary search on ε to determine a maximum number of failures that will be supported in the system for a given latency $\mathcal{L}$. The overall complexity of the algorithm remains polynomial, even though the running time is increased.

Next, we discuss the approach when both values of the latency and of the failure number are given. Our goal is then to detect the infeasibility of the combination before the end of the scheduling process. This would allow to reduce the latency or the number of supported failures during the execution of the scheduling algorithm, a nice feature when scheduling very large task graphs. To this purpose, we assign a deadline $d(t_i)$ to each task $t_i \in G$. The computation is done recursively in reverse topological order as follow:

$\forall\ t_i \in G$, if $\Gamma^+(t_i) = \emptyset$ then $d(t_i) \leftarrow \mathcal{L}$
else $\quad d(t_i) \leftarrow \min\limits_{t_j \in \Gamma^+(t_i)} \left\{ d(t_j) - \overline{\mathcal{E}(t_j)} - \overline{W(t_i,t_j)} \right\}$

---

where $\overline{\mathcal{E}(t_i)}$ and $\overline{W(t_i,t_j)}$ are respectively:
- the average execution time of t_i on the $\varepsilon+1$ fastest processors in the system. It is defined as $\overline{\mathcal{E}(t_i)} = \frac{\sum_{j=1}^{\varepsilon+1} \mathcal{E}(t_i,\mathcal{P}_j)}{\varepsilon+1}$,
- the average communication cost of the edge (t_i,t_j) defined as $\overline{W(t_i,t_j)} = \mathcal{V}(t_i,t_j).\overline{d}$, where $\overline{d}$ is the average delay to send a unit length data on the $\varepsilon+1$ fastest links in the system.

Deadlines are assigned so that a task deadline is always earlier than that of its successors. We can easily prove that, if at some step in the scheduling process, $\mathcal{P}^{(\varepsilon+1)}$ is the set of $\varepsilon+1$ processors that allow for the minimum finish time of task t, and $\max\limits_{\mathcal{P} \in \mathcal{P}^{(\varepsilon+1)}} \mathcal{F}(t,\mathcal{P}) > d(t)$, then both criteria cannot be satisfied simultaneously.

The scheduling scheme adopted when both criteria are fixed is similar to the FTSA algorithm. The only difference lies in the following test that we can add on line 10 in Algorithm 4.1, in order to check the feasibility of the criteria at each step of the algorithm:

If $\max_{\mathcal{P} \in \mathcal{P}^{(\varepsilon+1)}} \mathcal{F}(t, \mathcal{P}) \leq d(t)$ **then**
  Schedule t on the $\varepsilon + 1$ corresponding processors;
else
  return "Failed to satisfy both criteria simultaneously";

If at some step of the scheduling process, the algorithm fails to satisfy both criteria at the same time, the user can relax either ε or $\mathcal{L}$, and launch the algorithm again.

5 A Brief Description of FTBAR

In order to compare our algorithm to FTBAR [8], we give here a brief description of this algorithm, using the original notations of [8]. To the best of our knowledge, FTBAR is the only algorithm that addresses the same scheduling problem as this paper.

FTBAR (Fault Tolerance Based Active Replication) is based on an existing list scheduling algorithm presented in [27]. At each step n in the scheduling process, one free task is selected from the list based on the cost function $\sigma^{(n)}(t_i, p_j)$, called *schedule pressure*, it is computed as follows: $\sigma^{(n)}(t_i, p_j) = S^{(n)}(t_i, p_j) + \overline{s}(t_i) - R^{(n-1)}$. $S^{(n)}(t_i, p_j)$ is the earliest start-time (top-down) of t_i on p_j, similarly, $\overline{s}(t_i)$ is the latest start-time (bottom-up) of t_i and $R^{(n-1)}$ is the schedule length at step $n-1$. The selected task-processor pair is obtained as follows:
i) select for each free task t_i, the $\mathcal{N}pf + 1$ processor having the minimum *schedule pressure*
  $\cup_{l=1}^{l=\mathcal{N}pf+1} \sigma_{best}^{(n)}(t_i, p_{il}) \leftarrow \min_{p_j \in P}^{\mathcal{N}pf+1} \sigma^{(n)}(t_i, p_j)$.
ii) select the best pair among the previous set, *i.e.*, the one having the maximum *schedule pressure* (the most urgent pair) $\sigma_{urgent}^{(n)}(t) \leftarrow \max_{t_i \in freelist} \cup_{l=1}^{l=\mathcal{N}pf+1} \sigma_{best}^{(n)}(t_i, p_{il})$.

The task t is then scheduled on the $\mathcal{N}pf + 1$ processors computed at step 1. Ties are broken randomly. A recursive *Minimize-Start-Time* procedure proposed by Ahmad and Kwok [1] is used in attempting to reduce the start time of the selected

task t. The time complexity of the algorithm is $O(PN^3)$, where P is the number of processors in the system and N the number of tasks in G.

6 Experimental results

To evaluate the performance of FTSA, several series of simulations have been conducted. We use randomly generated graphs, whose parameters are consistent with those used in the literature [4, 8, 22]. The number of tasks is chosen uniformly from the range $[100, 150]$. The granularity of the task graph is varied from 0.2 to 2.0, with increments of 0.2. The number of processors is set to 20 and we let $\varepsilon = \{1, 2, 5\}$. To account for communication heterogeneity in the system, the unit message delay of the links and the message volume between two tasks are chosen uniformly from the ranges $[0.5, 1]$ and $[50, 150]$ respectively. Each point in the figures represents the mean of executions on 60 random graphs.

The metrics which characterize the performance of the algorithms are the latency and the overhead due to the active replication scheme. We compare the performances of FTSA and FTBAR. For each algorithm, we compare the fault free version (without replication) and the fault tolerant algorithm. Finally, MC-FTSA (Minimum Communications-FTSA) is the variant of FTSA which minimizes the amount of communications, using the greedy algorithm to select edges, as detailed in Section 4.2. Recall that the lower and upper bounds of the schedules are computed according to equations (2) and (4). The fault free schedule is defined as the schedule generated by each algorithm without replication, assuming that the system is completely safe.

Each algorithm is evaluated in terms of achieved latency and fault tolerance overhead. The latter is given by the following formula:

$$\text{Overhead} = \frac{\text{FTSA}^{\ell b}|\text{FTBAR}^{\ell b}|\text{FTSA}^{c}|\text{FTBAR}^{c} - \text{FTSA}^{*}}{\text{FTSA}^{*}}$$

where the superscripts ℓb, $*$ and c respectively denote the lower bound, the latency achieved by the fault free schedule, and the latency achieved by the schedule when processors effectively fail (crash).

Looking at figures plotting bounds (Figures 1(a), 2(a) and 3(a)), we see that FTSA achieves a really good lower bound, which is very close to the fault free version. As expected, the lower bound of MC-FTSA is slightly higher than that of FTSA, but its upper bound is close to the lower bound since we keep only the best communication edges in the schedule. The upper bound, which is a guaranty of the

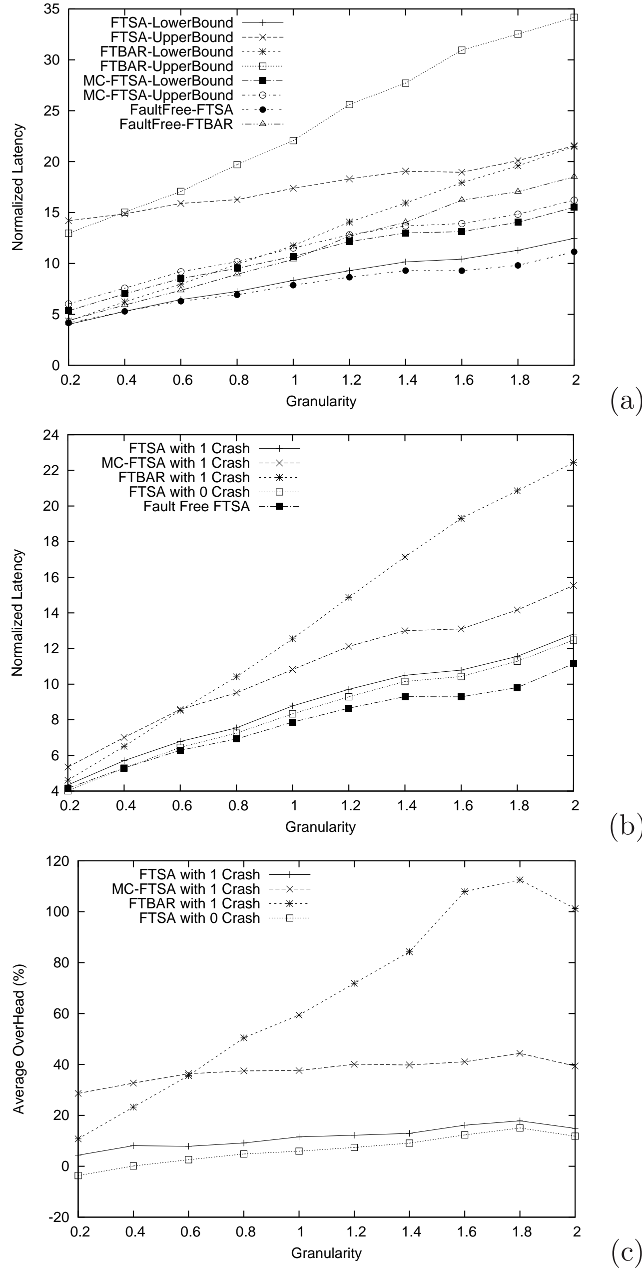

Figure 1: Average normalized latency and overhead comparison between FTSA, MC-FTSA and FTBAR (Bound and Crash cases, $\varepsilon = 1$)

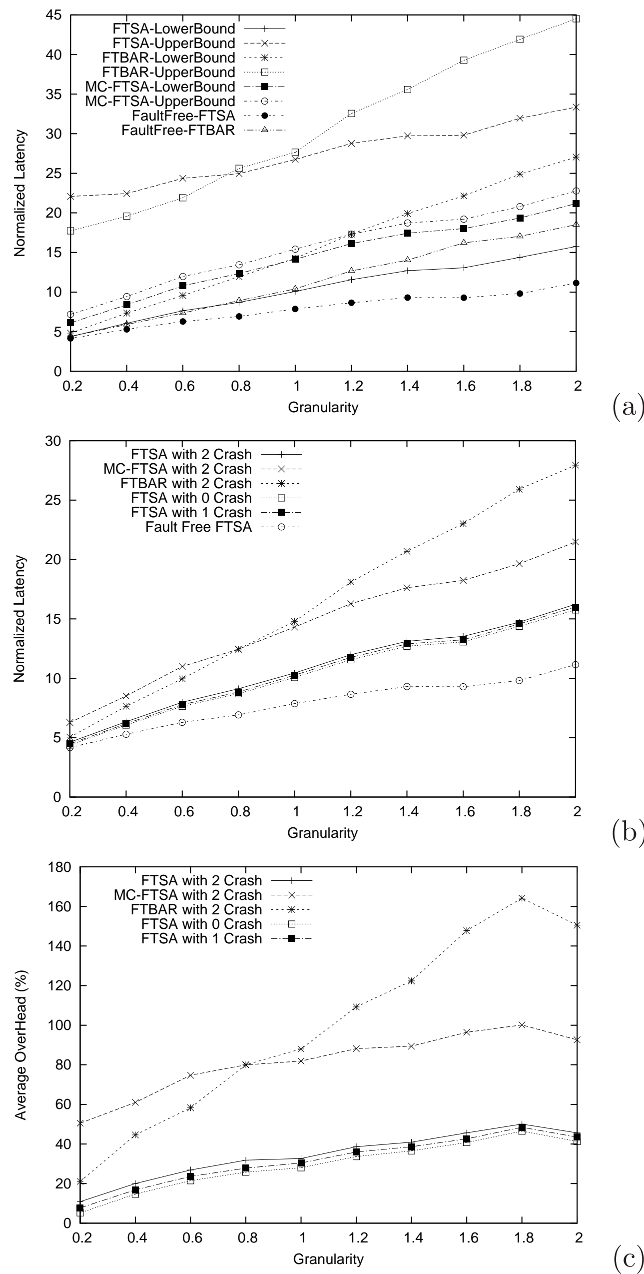

Figure 2: Average normalized latency and overhead comparison between FTSA, MC-FTSA and FTBAR (Bound and Crash cases, $\varepsilon = 2$)

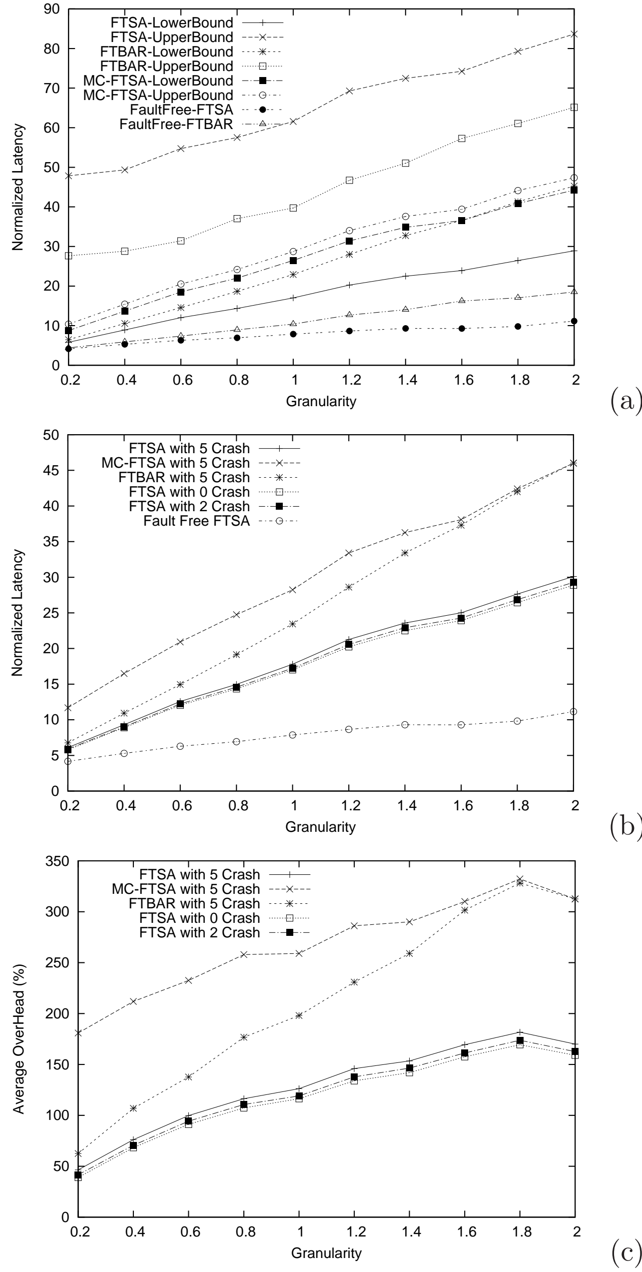

Figure 3: Average normalized latency and overhead comparison between FTSA, MC-FTSA and FTBAR (Bound and Crash cases, $\varepsilon = 5$)

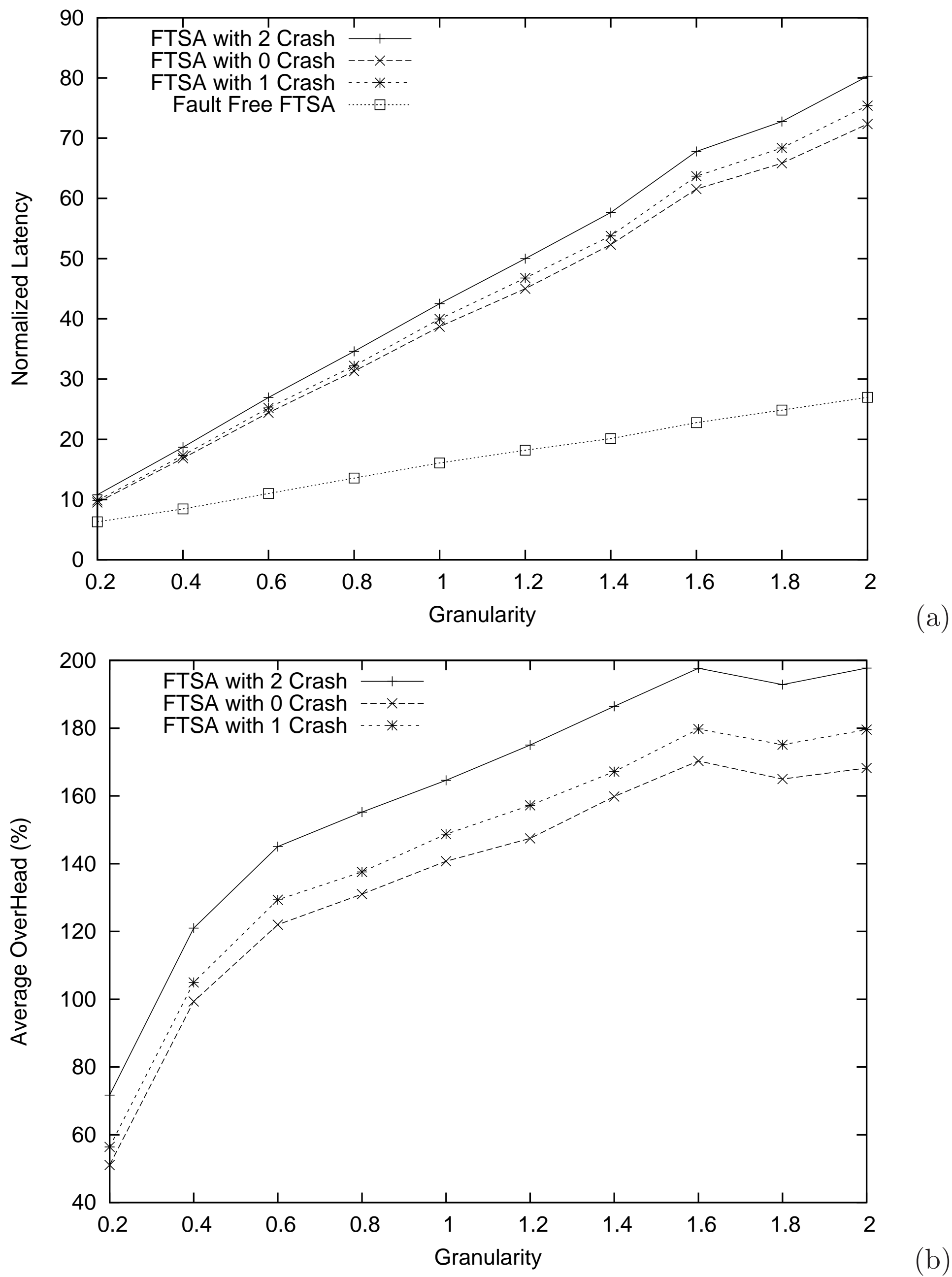

Figure 4: Average normalized latency and overhead comparison for FTSA with 0, 1 and 2 Crash (with 5 processors, $\varepsilon = 2$)

achieved latency, is even better than the lower bound of FTBAR for $\varepsilon = \{1, 2\}$ and a granularity greater than 1. FTSA always outperforms FTBAR in terms of lower bound. The reason of the poorer performance of FTBAR can be explained by the inconvenience of the schedule pressure function adopted for the processor selection. Processors are selected in such a way that the schedule pressure value is minimized. Doing so, tasks are not really mapped on those processors which would allow them to finish earlier.

We have also compared the behavior of each algorithm when processors crash down by computing the real execution time for a given schedule rather than just bounds. Processors that fail during the schedule process are chosen uniformly from the range $[1, 20]$. We can see on Figures 1(b), 2(b) and 3(b)) that FTSA^c behaves better than FTBAR^c. As expected, MC-FTSA has a bigger latency, since we removed some of the communication links. When crashes occur, this later algorithm is constrained to the use of some particular communication links. Even in this case, MC-FTSA achieves a better latency than FTBAR for $\varepsilon = \{1, 2\}$, which corresponds to a reasonable number of failures for an architecture of 20 processors.

We readily observe from Figures 1, 2 and 3 that we deal with two conflicting objectives. Indeed, the fault tolerance overhead increases together with the number of supported failures. We also see that latency increases together with granularity, as expected.

From Figures 1(b,c), 2(b,c) and 3(b,c), it is interesting to note that when the number of failures increases, there is not really much difference in the increase of the latency, compared to the schedule length generated with 0 crash (the lower bound). This is explained by the fact that the increase in the schedule length is already absorbed by the replication done previously, in order to resist to eventual failures. However on an architecture with fewer processors (for instance with 5 processors, see Figure 4), we clearly see the difference in terms of latency increase and therefore in terms of overhead, when the number of failures gets larger.

Finally, we realized some timing experiments to show the efficiency of the new heuristics in terms of execution time. The running times of FTSA, MC-FTSA and FTBAR are given in Table 1 for a case with 50 processors and 5 supported failures. The implementation is in C and experiments are run on a Core 2 Duo processor (CPU 1.66 GHz). From this table, we observe that our algorithms FTSA and MC-FTSA are considerably faster than FTBAR. We conclude that for large task graphs, the FTBAR algorithm is not really practical, while FTSA and MC-FTSA are still capable of scheduling very large task graphs in reasonable time.

Table 1: Running Times in seconds

Number of tasks	FTSA	MC-FTSA	FTBAR
100	0.01	0.02	0.15
500	0.08	0.12	4.19
1000	0.16	0.24	17.10
2000	0.30	0.50	71.22
3000	0.46	0.75	167.57
5000	0.77	1.28	465.75

7 Conclusion

In this paper we have presented FTSA, an efficient fault-tolerant scheduling algorithm for heterogeneous systems based on an active replication scheme. We have also designed MC-FTSA, a variant of FTSA in which the communication overhead due to task replication is dramatically reduced. To assess the performance of FTSA, simulation studies were conducted to compare it with FTBAR, which seems to be its only direct competitor from the literature. We have shown that FTSA is superior to FTBAR both in terms of computational complexity and quality of the resulting schedule. We also point out that MC-FTSA generates better schedules than FTBAR when there is a small number of failures.

We plan to investigate more realistic communication models such as the bounded multi-port model [13] or the one-port model [25]. Such models are more realistic because they bound the volume of data that can be sent by a given processor (due to the limited capacity of its network card), as well as the volume of data that can share a given communication link (due to the limited bandwidth of the link). With these models, we expect MC-FTSA to be superior to other scheduling algorithms, since it already accounts for reduced communications. Also, we want to study a more complex failure model, in which we would also account for the failure probability of the application.

References

[1] Ishfaq Ahmad and Yu-Kwong Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*,

1998.

[2] R. Al-Omari, Arun K. Somani, and G. Manimaran. Efficient overloading techniques for primary-backup scheduling in real-time systems. *Journal of Parallel and Distributed Computing*, 64(5):629–648, 2004.

[3] Olivier Beaumont, Vincent Boudet, and Yves Robert. The iso-level scheduling heuristic for heterogeneous processors. In *PDP*, pages 335–342, 2002.

[4] Atakan Dogan and Fusun Ozguner. Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(03):308–323, 2002.

[5] Jack Dongarra, Emmanuel Jeannot, Erik Saule, and Zhiao Shi. Bi-objective scheduling algorithms for optimizing makespan and reliability on heterogeneous systems. In *Proc. of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'07*, pages 280–288, 2007.

[6] Apostolos Gerasoulis and Tao Yang. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.

[7] Sunondo Ghosh, Rami Melhem, and Daniel Mosse. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):272–284, 1997.

[8] A. Girault, H. Kalla, M. Sighireanu, and Y. Sorel. An algorithm for automatically obtaining distributed and fault-tolerant static schedules. In *International Conference on Dependable Systems and Networks, DSN'03*, 2003.

[9] Mourad Hakem and Franck Butelle. Critical path scheduling parallel programs on unbounded number of processors. *International Journal of Foundations of Computer Science*, 17(2):287–301, 2006.

[10] Mourad Hakem and Franck Butelle. Reliability and scheduling on systems subject to failures. In *Proc. of the 36th IEEE International Conference on Parallel Processing ICPP'07*, page 38, 2007.

[11] K. Hashimito, T. Tsuchiya, and T. Kikuno. A new approach to realizing fault-tolerant multiprocessor scheduling by exploiting implicit redundancy. In *Proc.*

of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97), page 174, 1997.

[12] K. Hashimito, T. Tsuchiya, and T. Kikuno. Effective scheduling of duplicated tasks for fault-tolerance in multiprocessor systems. *IEICE Transactions on Information and Systems*, E85-D(3):525–534, 2002.

[13] B. Hong and V.K. Prasanna. Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. In *International Parallel and Distributed Processing Symposium IPDPS'2004*. IEEE Computer Society Press, 2004.

[14] Chao-Ju Hou and Kang G. Shin. Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 46(12):1338–1356, 1997.

[15] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, 1989.

[16] Santhanam Srinivasan Niraj K. Jha. Safety and reliability driven task allocation in distributed systems. *IEE Trans. on Parallel and Dist. Syst.*, 10(03):238–251, 1999.

[17] S. Kartik and C. Siva Ram Murthy. Task allocation algorithms for maximizing reliability of distributed computing systems. *IEEE Trans. on Computers*, 41(06):719–724, 1997.

[18] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, 1996.

[19] G. Manimaran and C. Siva Ram Murthy. A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1137–1152, 1998.

[20] Martin Naedele. Fault-tolerant real-time scheduling under execution time constraints. In *Proc. of the Sixth International Conference on Real-Time Computing Systems and Applications*, page 392, 1999.

[21] Xiao Qin and Hong Jiang. A dynamic and reliability driven scheduling algorithm for parallel real-time jobs executing on heterogeneous clusters. *Journal of Parallel and Distributed Computing*, 65(08):885–900, 2005.

[22] Xiao Qin and Hong Jiang. A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems. *Parallel Computing*, 32(5):331–346, 2006.

[23] V. Sarkar. *Partitionning and Scheduling Parallel Programs for Execution on Multiprocessors*. MIT Press, 1989.

[24] G.C. Sih and E.A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. on Parallel and Dist. Systems*, 4(2):75–87, 1993.

[25] O. Sinnen and L. Sousa. Scheduling task graphs on arbitrary processor architectures considering contention. In *High Performance Computing and Networking*, pages 373–382. Springer-Verlag LNCS 2110, 2001.

[26] Jia Ping Wang Sol M. Shatz and Masanori Goto. Task allocation for maximizing reliability of distributed computer systems. *IEEE Trans. on Computers*, 41(09):1156–1168, 1992.

[27] Yves Sorel. Massively parallel computing systems with real-time constraints: the "algorithm architecture adequation". In *Proc. of Massively Parallel Comput. Syst., MPCS*, 1994.

[28] Y.Oh and S.H.Son. Scheduling real-time tasks for dependability. *Journal of Operational Research Society*, 48(6):629–639, 1997.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399