Realistic models and efficient algorithms for fault-tolerant scheduling on heterogeneous platforms

Anne Benoit, Mourad Hakem, Yves Robert

▶ **To cite this version:**

Anne Benoit, Mourad Hakem, Yves Robert. Realistic models and efficient algorithms for fault-tolerant scheduling on heterogeneous platforms. [Research Report] LIP RR-2008-09, Laboratoire de l'informatique du parallélisme. 2008, 2+21p. hal-02102653

HAL Id: hal-02102653

https://hal-lara.archives-ouvertes.fr/hal-02102653v1

Submitted on 17 Apr 2019

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.

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

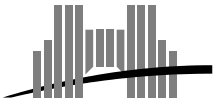
Realistic Models and Efficient Algorithms for Fault Tolerant Scheduling on Heterogeneous Platforms

Anne Benoit,
Mourad Hakem,
Yves Robert

February 2008

Research Report N° 2008-09

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`



Realistic Models and Efficient Algorithms for Fault Tolerant Scheduling on Heterogeneous Platforms

Anne Benoit, Mourad Hakem, Yves Robert

February 2008

Abstract

Most list scheduling heuristics rely on a simple platform model where communication contention is not taken into account. In addition, it is generally assumed that processors in the systems are completely safe. To schedule precedence graphs in a more realistic framework, we introduce an efficient fault tolerant scheduling algorithm that is both contention-aware and capable of supporting ε arbitrary fail-silent (fail-stop) processor failures. We focus on a bi-criteria approach, where we aim at minimizing the total execution time, or latency, given a fixed number of failures supported in the system. Our algorithm has a low time complexity, and drastically reduces the number of additional communications induced by the replication mechanism. Experimental results fully demonstrate the usefulness of the proposed algorithm, which leads to efficient execution schemes while guaranteeing a prescribed level of fault tolerance.

Keywords: Communication contention, fault tolerance, multi-criteria scheduling, heterogeneous systems.

1 Introduction

The efficient scheduling of application tasks is critical in order to achieve high performance in parallel and distributed systems. The objective of scheduling is to find a mapping of the tasks onto the processors, and to order the execution of the tasks so that: (i) task precedence constraints are satisfied; and (ii) a minimum schedule length is provided.

Task graph scheduling is usually studied using the so-called *macro-dataflow* model, which is widely used in the scheduling literature: see the survey papers [20, 23, 7, 8] and the references therein. This model was introduced for homogeneous processors, and has been (straightforwardly) extended for heterogeneous computing resources. In a word, there is a limited number of computing resources, or processors, to execute the tasks. Communication delays are taken into account as follows: let task t be a predecessor of task t' in the task graph; if both tasks are assigned to the same processor, no communication overhead is paid, the execution of t' can start right at the end of the execution of t; on the contrary, if t and t' are assigned to two different processors P and P', a communication delay is paid. More precisely, if P finishes the execution of t at time-step x, then P' cannot start the execution of t' before time-step $x + \mathsf{comm}(t, t', P, P')$, where $\mathsf{comm}(t, t', P, P')$ is the communication delay (which depends upon both tasks t and t' and both processors P and P'). Because memory accesses are typically one order of magnitude cheaper than inter-processor communications, it makes good sense to neglect them when t and t' are assigned to the same processor.

However, the major flaw of the macro-dataflow model is that communication resources are not limited. First, a processor can send (or receive) an arbitrary number of messages in parallel, hence an unlimited number of communication ports is assumed (this explains the name *macro-dataflow* for the model). Second, the number of messages that can simultaneously circulate between processors is not bounded, hence an unlimited number of communications can simultaneously take place on a given link. In other words, the communication network is assumed to be contention-free, which of course is not realistic as soon as the processor number exceeds a few units.

We strongly believe that the macro-dataflow task graph scheduling model should be modified to take communication resources into account. Recent papers [13, 15, 24, 25] made a similar statement and introduced variants of the model (see the discussion of related work in Section 3). In this paper, we suggest to use the bi-directional one-port architectural model, where each processor can communicate (send and/or receive) with at most one other processor at a given time-step. In other words, a given processor can simultaneously send a message, receive another message, and perform some (independent) computation.

There is yet another reason to revisit traditional list scheduling techniques. With the advent of large-scale heterogeneous platforms such as clusters and grids, resource failures (processors/links) are more likely to occur and have an adverse effect on the applications. Consequently, there is an increasing need for developing techniques to achieve fault tolerance, *i.e.*, to tolerate an arbitrary number of failures during execution. Scheduling for heterogeneous platforms 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 Contention-Aware Fault Tolerant (CAFT) scheduling algorithm that aims at tolerating multiple processor failures without sacrificing the latency. CAFT 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. Experimental results demonstrate that our algorithm outperforms other algorithms that were initially designed for the macro-dataflow model, such as FTSA [4], a fault tolerant extension of HEFT [27], and FTBAR [10].

The rest of the paper is organized as follows: Section 2 presents basic definitions and assumptions. We overview related work in Section 3. Then we outline the principle of FTSA [4] and FTBAR [10] as well as their adaptation to the one-port model in Section 4. Section 5 describes the new CAFT algorithm. We experimentally compare CAFT with FTSA and FTBAR in Section 6; the results assess the very good behavior of our algorithm. 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 target parallel heterogeneous system consists of a finite number of processors $P = \{P_1, P_2, \ldots, P_m\}$ connected by a dedicated communication network. The processors are fully connected, i.e, every processor can communicate with every other processor in the system. The link between processors P_k and P_h is denoted by $l_{P_k P_h}$. As stated above, in the traditional macro-dataflow model, there is no contention for communications ressources, and an unlimited number of communications can be executed concurrently. The bi-directional one-port model considered in this work deals with communication contention as follows:

- At a given time-step, any processor can send a message to, and receive a message from, at most one other processor. Network interfaces are assumed full-duplex in this bidirectional model.
- Communication and (independent) computation may fully overlap. As in the traditional model, a processor can execute at most one task at each time-step.
- Communications that involve disjoint pairs of sending/receiving processors can occur in parallel.

Several variants could be considered, such as uni-directional communications, or no communication/computation overlap. But the bi-directional one-port model seems closer to the actual capabilities of modern networks (see the discussion in Section 3).

The computational heterogeneity of tasks is modeled by a function $\mathcal{E} : V \times \mathcal{P} \rightarrow R^+$, which represents the execution time of each task on each processor in the system: $\mathcal{E}(t, P_k)$ denotes the execution time of t on P_k, $1 \leq k \leq m$. The heterogeneity in terms of communications is expressed by $W(t_i, t_j) = \mathcal{V}(t_i, t_j).d(P_k, P_h)$, where task t_i is mapped on processor P_k, task

t_j is mapped on processor P_h, and $d(P_k, P_h)$ is the time required to send a unit length data from P_k to P_h. The communication has no cost if two tasks in precedence are mapped on the same processor: $d(P_k, P_k) = 0$.

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 of the DAG G on the platform $\mathcal{P}$ obeying the one-port model. The objective is to minimize the latency $\mathcal{L}(G)$, while tolerating an arbitrary number ε of processor failures. 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.

3 Related work

Contention-aware task scheduling is considered only in a few papers from the literature [13, 15, 24, 25]. In particular, Sinnen and Sousa [24, 25] show through simulations that taking contention into account is essential for the generation of accurate schedules. They investigate both end-point and network contention. Here end-point contention refers to the bounded multi-port model [14]: the volume of messages that can be sent/received by a given processor is bounded by the limited capacity of its network card. Network contention refers to the one-port model, which has been advocated by [5, 6] because "current hardware and software do not easily enable multiple messages to be transmitted simultaneously." Even if non-blocking multi-threaded communication libraries allow for initiating multiple send and receive operations, all these operations "are eventually serialized by the single hardware port to the network." Experimental evidence of this fact has recently been reported by Saif and Parashar [22], who report that asynchronous sends become serialized as soon as message sizes exceed a few megabytes. Their results hold for two popular implementations of the MPI message-passing standard, MPICH on Linux clusters and IBM MPI on the SP2. The one-port model fully accounts for the heterogeneity of the platform, as each link has a different bandwidth. It generalizes a simpler model studied by Banikazemi [3], Liu [17], and Khuller and Kim [16]. In this simpler model, the communication time only depends on the sender, not on the receiver: in other words, the communication speed from a processor to all its neighbors is the same.

All previous scheduling heuristics were developed for minimizing latency on realistic parallel platform models, assuming that processors in the system are completely safe. i.e, they do not deal with fault tolerance.

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, 9, 10, 11, 12, 18, 19, 21, 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 [19, 28]. To achieve high schedulability while providing fault-tolerance, the heuristics presented in [2, 9, 18, 21] 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.

All algorithms belonging to this class [2, 9, 18, 19, 21, 28] share two common points: (i) only one processor can fail at any time, and a second processor cannot fail before the system recovers from the first failure; and (ii) they are designed for the macro-dataflow model.

(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 [10]. Recently, we have proposed the FTSA algorithm [4], a fault tolerant extension of HEFT [27]. We showed that FTSA outperforms FTBAR in terms of time complexity and solution quality. Here again, FTSA and FTBAR have been designed for the macro-dataflow model. A brief description of FTSA and FTBAR, as well as their adaptation to the one-port model, is given in Section 4.

To summarize, all previous fault-tolerant algorithms assume no restriction on communication ressources, which makes them impractical for real life applications. To the best of our knowledge, the work presented in this paper is the first to tackle the combination of contention awareness and fault tolerance scheduling.

4 Fault-tolerant heuristics

In this section, we briefly outline the the main features of FTBAR [10] and FTSA [4], that both were originally designed for the macro-datflow model. Next we show how to modify them for an execution under the one-port model.

4.1 FTBAR

FTBAR [10] (Fault Tolerance Based Active Replication) is based on an existing list scheduling algorithm presented in [26]. Using the original notations of [10], 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.

4.2 FTSA

FTSA (Fault Tolerant Scheduling Algorithm) has been introduced in [4] as a fault-tolerant extension of HEFT [27]. At each step of the mapping process, FTSA selects a free task t (a task is free if it is unscheduled and if all of its predecessors are scheduled) with the highest priority and simulates its mapping on all processors. The first $\varepsilon+1$ processors that allow the *minimum finish time* of t are kept. The finish time of a task t on processor P depends on the time when at least one replica of each predecessor of task t has sent its results to P (and, of course, processor P must be ready). Once this set of processors is determined, the task t is scheduled on these $\varepsilon+1$ distinct processors (replicas). The latency of the schedule is the latest time at which at least one replica of each task has been computed, thus it represents a **lower bound** (this latency can be achieved if no processor permanently fails during execution). The **upper bound**, always achieved even with ε failures, is computed using as a finish time the completion time of the last replica of a task (instead of the first one for the lower bound). A formal definition can be found in [4]. The time complexity of the algorithm is $O(em^2+v\log\omega)$, where ω is the *width* of the task graph (the maximum number of tasks that are independent in G). Recall that v is the number of tasks, and e the number of edges in G.

Note that with FTSA, each task of the task graph G is replicated $\varepsilon+1$ times, because duplicating each task $\varepsilon+1$ times is an absolute requirement to resist to ε failures. 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. Still, the total number of communications induced by the fault-tolerant mechanism is very large. The same comment applies to FTBAR, where each replica of a task communicates data to each replica of its successors.

4.3 Adaptation to the one-port model

In order to adapt both FTSA and FTBAR algorithms to the one-port model, we have to take constraints related to communication resources into account. The idea consists in serializing incoming and outgoing communications on the links.

A communication c on link l is characterized by its start time $\mathcal{S}(c,l)$ and its finish time $\mathcal{F}(c,l)$. Also, we define $\mathcal{R}(l)$ as the ready time of a communication link l: $\mathcal{R}(l) = \max_{c_k \text{ on } l}\Big(\mathcal{F}(c_k,l)\Big)$. In the following, we formalize all the one-port constraints:

i) **Link constraint**: For any two communications c, c' scheduled on link l,

$$\mathcal{F}(c,l) \leq \mathcal{S}(c',l) \quad \vee \quad \mathcal{F}(c',l) \leq \mathcal{S}(c,l) \tag{1}$$

Inequality (1) states that any two communications c and c' do not overlap on a link.

ii) **Sending constraint**: For any two communications $c_{ij}, c_{ij'}$ sent from a given processor P_i to two processors $\mathcal{P}_j$, $P_{j'}$,

$$\mathcal{F}(c_{ij},l_{ij}) \leq \mathcal{S}(c_{ij'},l_{ij'}) \vee \mathcal{F}(c_{ij'},l_{ij'}) \leq \mathcal{S}(c_{ij},l_{ij}) \tag{2}$$

iii) **Receiving constraint**: For any two communications $c_{ji}, c_{j'i}$ sent from processors P_j and $P_{j'}$ to the same processor P_i,

$$\mathcal{S}(c_{ji},l_{ji}) \geq \mathcal{F}(c_{j'i},l_{j'i}) \vee \mathcal{S}(c_{j'i},l_{j'i}) \geq \mathcal{F}(c_{ji},l_{ji}) \tag{3}$$

Inequalities (2) and (3) ensure that any two incoming/outgoing communications c and c' must be serialized at their reception/emission site.

Let t_i be a task scheduled on processor P. Let $\mathcal{S}_F(P)$ be the sending free time of P, i.e, the time on which the communication $c_{ij}, 1 \leq j \leq |\Gamma^+(t_i)|$, can start from processor P. The earliest start time of the communication c_{ij} scheduled to the link l and its finish time are defined by the following equations:

$$\begin{aligned} &\mathcal{S}(c_{ij},l) = \max\Big(\mathcal{S}_F(P), \mathcal{F}\Big(t_i,P\Big), \mathcal{R}(l)\Big), \\ &\mathcal{F}(c_{ij},l) = \mathcal{S}(c_{ij},l) + W(c_{ij},l) \end{aligned} \tag{4}$$

Thus, communication c_{ij} is constrained by both $\mathcal{S}_F(P)$, $\mathcal{R}(l)$ and the finish time of its source task t_i on P. It can start as soon as the processing of the task is finished only if we have $\mathcal{F}(t_i,P) \geq \mathcal{S}_F(P)$ and $\mathcal{F}(t_i,P) \geq \mathcal{R}(l)$.

The start time of task t_i on processor P is constrained by communications incoming from its predecessors that are assigned on other processors. Thus, $\mathcal{S}(t_i,P)$ satisfies the following conditions: it is later than the time when all messages from t_i's predecessors arrive on processor P, and it is later than the ready time of processor P, defined as the maximum of finish time of all tasks already scheduled on P. Let $\mathcal{A}(c,P)$ be the time when communication c arrives on processor $\mathcal{P}$, and $r(P)$ be the ready time of P. The start time of t_i on P is defined as follows:

$$\mathcal{S}(t_i,P) = \max\Big(\max_{t_j \in \Gamma^-(t_i)}\big\{\mathcal{A}(c_{ji},P)\big\}, r\big(P\big)\Big) \tag{5}$$

where c_{ji} is the communication from t_j to t_i.

The arrival time $\mathcal{A}(c_{ji},P)$ is computed for each predecessor as follows. Let l_j be the communication link that connects the processor on which t_j is mapped to P. Let $\mathcal{R}_F(P)$ be the receiving free time of P, i.e, the time when P is ready to receive data. We sort predecessors $t_j \in \Gamma^-(t_i)$ by non-decreasing order of their communication finish time $\mathcal{F}(c_{ji},l_j)$, and renumber them from 1 to $|\Gamma^-(t_i)|$.

$$\begin{aligned} &\forall \;\; 1 \leq j \leq |\Gamma^-(t)|, \quad \mathcal{A}(c_{ji},P) \leftarrow W(c_{ji},l_j) + \\ &\max\big(\mathcal{R}_F(P), \mathcal{F}(c_{(j-1)i},l_{j-1}), \mathcal{F}(c_{ji},l_j) - W(c_{ji},l_j)\big) \\ &\text{with} \;\; \mathcal{F}(c_{0i},l_0) = 0 \end{aligned} \tag{6}$$

Equation (6) shows that the arrival time $\mathcal{A}(c, P)$ is constrained by the receiving free time $\mathcal{R}_F(P)$ of P. In addition, it complies with the inequality (3), i.e, concurrent communications are serialized at the reception site.

5 CAFT scheduling algorithm

The one-port model enforces to serialize communications. But as pointed out above, the duplication mechanism induces a large number of additional communications. Therefore, we expect execution time to dramatically increase together with the number of supported failures. This calls for the design of a variant of FTSA where the number of communications induced by the replication scheme is drastically reduced. The main idea of the new CAFT (Contention-Aware Fault Tolerant) scheduling algorithm is to have each replica of a task communicate to a unique replica of its successors whenever possible, while preserving the fault tolerance capability (guaranteeing success if at most ε processors fail during execution). Communicating to a single replica is only possible in special cases, typically for tasks having a unique predecessor, or when every replica of the several predecessors are all mapped onto distinct processors. When these constraints are not satisfied, we greedily add extra communications to guarantee failure tolerance, as illustrated below through a small example.

In the following, we denote by $\mathcal{B}(t)$ the set of $\varepsilon+1$ replicas of a task t. Also, we denote by $t^{(k)}$ those replicas, for $1 \leq k \leq \varepsilon+1$. Thus, $\mathcal{B}(t) = \{t^{(1)}, ..., t^{(\varepsilon+1)}\}$. $P(t^{(k)})$ is the processor on which this replica is scheduled.

Let t_i be the current task to be scheduled by CAFT. Consider a predecessor t_j of t_i, $j \in \Gamma^-(t_i)$, that has been replicated on $\varepsilon+1$ distinct processors. We aim at orchestrating communications incoming from predecessors t_j to t_i so that each replica in $\mathcal{B}(t_j)$ communicates to only one replica in $\mathcal{B}(t_i)$ when possible, rather than communicating to all of them as in the FTSA and FTBAR algorithms.

If t_i has only one predecessor t_1, then a one-to-one communication scheme resists to ε failures, as it was proved in [4]. We find the best mapping in which each $t_1^{(k)}$ sends data to exactly one of the $t_i^{(k_i)}$. The problem becomes more complex when t_i has more than one predecessor, for instance t_1, t_2 and t_3, and replicas of different instances are mapped on a same processor. For instance, let us have $\varepsilon = 1$, $t_1^{(1)}$ and $t_2^{(1)}$ mapped on P_1, $t_1^{(2)}$ and $t_3^{(1)}$ mapped on P_2, $t_2^{(2)}$ and $t_3^{(2)}$ mapped on P_3. In all possible schedulings for t_i, both $t_i^{(1)}$ and $t_i^{(2)}$ need to receive data from two distinct processors. One processor between P_1, P_2 and P_3 must thus communicate with both replicas. If this particular processor crashes, both replicas will miss data to continue execution, and thus the application cannot tolerate this single failure. In such cases in which a processor is processing several replicas of predecessors and communicating with different replicas of t_i, we need to add extra communications to ensure failure tolerance.

Algorithm 5.1 is the main CAFT algorithm. Tasks are scheduled in an order defined by the priority of the task: 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 top level and the bottom level of task t. The top level $t\ell(t)$ is the length of the longest path from an entry (top) node to t (excluding the execution time of t) in the current partially clustered DAG. The top level of an entry node is zero. Top levels are computed according to a traversal of the graph in topological order. The bottom level $b\ell(t)$ is the length of the longest path starting at t to an exit node in the graph. The bottom level of an exit node is equal to its execution time. Bottom levels are computed according to a traversal of the graph in reverse topological order. Note that path lengths are defined

as the average sum of edge weights and node weights (see [27, 4]). $\mathcal{H}(\ell)$ is the head function which returns the first replica/task from a sorted list ℓ, where the list is sorted according to replicas/tasks priorities (ties are broken randomly). The difficult point consists in deciding where to place current task t in order to minimize the amount of communications. Also, communications should be orchestrated to avoid useless data transfer between replicas.

Let us define a *singleton processor*, as a processor with only one instance/replica $t_j^{(k)}$, $1 \leq j \leq |\Gamma^-(t_i)|$, $1 \leq k \leq \varepsilon + 1$ and $\mathcal{X} \subseteq \bigcup_{j=1}^{j=|\Gamma^-(t_i)|} \{P(\mathcal{B}(t_j))\}$ be the set of such *singleton processors*. Let $\overline{\mathcal{B}}(t_j)$ be the subset of replicas of each predecessor t_j scheduled in $\mathcal{X}$ and $\lambda_j = |\overline{\mathcal{B}}(t_j)|$. Let T be a subset of replicas selected from the set $\bigcup_{j=1}^{j=|\Gamma^-(t_i)|} \{\overline{\mathcal{B}}(t_j)\}$.

Algorithm 5.1 The CAFT Algorithm

$P = \{P_1, P_2, \dots P_m\}$; *(*Set of processors*)*
$\varepsilon \leftarrow$ maximum number of supported failures
$\mathbb{P} = \emptyset$;
Compute $b\ell(t)$ for each task t in G and set $t\ell(t) = 0$ for each entry task t;
$S = \emptyset$; $U = V$; *(*Mark all tasks as unscheduled*)*
$\alpha = \emptyset$; *(*list of free tasks*)*
Put entry tasks in α;
while $U \neq \emptyset$ **do**
  $t \leftarrow \mathcal{H}(\alpha)$; *(*Select task with highest priority *)*
  $\forall$ $1 \leq j \leq |\Gamma^-(t_i)|$, compute λ_j;
  $\theta \leftarrow \min_j(\lambda_j)$; $i = 0$;
  while $i < \theta$ **do**
    One-To-One-Mapping(t);
    $i = i + 1$;
  end while
  while $\theta < \varepsilon + 1$ **do**
    Compute $\mathcal{F}(t, P_k)$ for $1 \leq k \leq m$ and $P_k \notin \mathbb{P}$ using equation (6);
    Keep the (task,processor) pair that allows the minimum finish time of t;
    $\theta = \theta + 1$;
  end while
  Put t in S and update priority values of t's successors;
  Put t's free successors in α;
  $U \leftarrow U \backslash \{t\}$;
end while

When there are enough singleton processors with replicas of predecessor tasks, we use the one-to-one mapping procedure described in Algorithm 5.2. This name stems from the fact that each replica in $\bigcup_{j=1}^{j=|\Gamma^-(t_i)|} \overline{\mathcal{B}}(t_j)$ should communicate to exactly one replica in $\mathcal{B}(t_i)$. The number of times the one-to-one-mapping procedure can be called for scheduling the $\varepsilon + 1$ replicas of the current task is determined by $\theta \leftarrow \min_j(\lambda_j)$. In this procedure, we denote by $\mathbb{P} \subseteq P$ the subset of "locked" processors which are already either involved in a communication with a replica of t_i, or processing it (i.e., the execution of a replica of $\mathcal{B}(t_i)$ has been scheduled on such a processor).

The computation of the finish time of t_i is simulated m times, once for every processor.

Algorithm 5.2 One-To-One-Mapping(t_i)

1: $k = 0$;
2: **while** $k \leq m$ and $P_k \notin \mathbb{P}$ **do**
3: $\forall\ 1 \leq j \leq |\Gamma^-(t_i)|$, sort the set $\overline{\mathcal{B}}(t_j)$ by non decreasing order of their communication finish time $\mathcal{F}(c, l)$ on the links;
4: $T \leftarrow \bigcup_{1 \leq j\ \leq |\Gamma^-(t_i)|} \mathcal{H}\Big(\overline{\mathcal{B}}(t_j)\Big)$;
5: Simulate the mapping of t_i on processor P_k as well as the communications induced by the replicas of the set T to the links;
6: $k = k + 1$;
7: **end while**
8: Select the (task, processor) pair that allows the earliest finish time of t_i as computed by equation (6);
9: Schedule t_i onto the corresponding processor (let's call it P^*) and the incoming communications to the corresponding links;
10: Update the set $\mathbb{P}$

$$\mathbb{P} \leftarrow \mathbb{P} \bigcup P^* \bigcup \left\{ \bigcup_{j=1}^{j=|\Gamma^-(t_i)|} P\left(\mathcal{H}\Big(\overline{\mathcal{B}}(t_j)\Big)\right) \right\} \tag{7}$$

11: Update each sorted list $\overline{\mathcal{B}}(t)$;
$\forall\ 1 \leq j \leq |\Gamma^-(t_i)|,\ \ \overline{\mathcal{B}}(t_j) \leftarrow \overline{\mathcal{B}}(t_j) \setminus \mathcal{H}\Big(\overline{\mathcal{B}}(t_j)\Big)$

Hence the mapping of each incoming communications onto the links is also simulated m times. To obtain an accurate view of the communications finish time on their respective links and the contention, the incoming communications are removed from the links before the procedure is repeated on the next processor.

In general, we cannot give an analytical expression of the actual number of communications induced by the CAFT algorithm. Still, we can bound the number of communications induced by CAFT for special graphs:

Proposition 5.1 *The total number of messages generated by CAFT for Fork/Outforest graphs is at most* $e(\varepsilon + 1)$.

Proof: An outforest graph is a directed graph in which the indegree of every task t in G is at most one $|\Gamma^-(t)| = 1$. To resist to ε failures, each task $t \in G$ should be replicated $\varepsilon + 1$ times. Therefore, at each step of the mapping process we have $\cap P\big(\mathcal{B}(t_* \prec t)\big) = \emptyset$ and $|\mathcal{X}| = \theta = \varepsilon + 1$. Thus, the *one-to-one mapping procedure* is performed $\theta = \varepsilon + 1$ times. This ensures that each replica $t_*^{(k)}$, $1 \leq k \leq \varepsilon + 1$ sends its data results to one and only one replica of each successor task. Therefore, each task $t \in G$ will receive its input data $|\Gamma^-(t)| = 1$ times. However, in some cases, we may have an intra-processor communication, when two replicas of two tasks in precedence are mapped onto the same processor. Thus, summing up for all the v tasks in G, the total number of messages is at most $\sum_{i=1}^{v} |\Gamma^-(t)|(\varepsilon + 1) = (v - 1)(\varepsilon + 1) = e(\varepsilon + 1)$. □

For general graphs, the number of communications will also be bounded by $e(\varepsilon + 1)$ if at each step replicas are assigned to different processors (same proof as above). This condition is not guaranteed to hold, and we will have to greedily add some additional communications to guarantee the robustness of CAFT. However, practical experiments (see Section 6) show

that CAFT always drastically reduces the total number of messages as compared to FTBAR or FTSA, thereby achieving much better performance.

Proposition 5.2 *The schedule generated by the CAFT algorithm is valid and resists to ε failures.*

When a current task t to be mapped has more than one predecessor and $\theta = 0$, the *one-to-one mapping procedure* is not executed and therefore CAFT algorithm performs more than $e(\varepsilon+1)$ communications. In this case we can resist to ε failures as it was proved in [4]. Therefore, we just need to check if the mapping of the θ replicas performed by the *one-to-one mapping procedure* resists to $\theta - 1$ failures.

Proof: The proof is composed of two parts:

i) *Deadlock/Mutual exclusion*: First we prove that we never fall into a deadlock trap as described by the example below. Consider a simple task graph composed of two tasks in precedence $t_1 \prec t_2$. Assume that $\varepsilon = 1$, $\mathcal{B}(t_1) = \{t_1^{(1)}, t_1^{(2)}\}$ whith $P\big(\mathcal{B}(t_1)\big) = \{P_1, P_2\}$ and $\mathcal{B}(t_2) = \{t_2^{(1)}, t_2^{(2)}\}$ whith $P\big(\mathcal{B}(t_2)\big) = \{P_1, P_3\}$. If we retain the communications $P_1(t_1^{(1)}) \rightarrow P_3(t_2^{(2)})$ and $P_2(t_1^{(2)}) \rightarrow P_1(t_2^{(1)})$, then the algorithm is blocked by the failure of P_1. But if we enforce that the only edge from P_1 goes to itself, then we resist to 1 failure.

Mutual exclusion is guaranteed by equation (7)(Algorithm 5.2, line 10). Indeed, by simulating the mapping of t_2 on P_1, P_2 and P_3, we have two possible scenarios:
1) - The first replica $t_2^{(1)}$ is mapped either on P_1 or on P_2, and in either cases the one assigned to the replica will be locked by equation (7). Suppose for instance that P_1 was chosen/locked, thus, to resist to 1 failure, the second replica $t_2^{(2)}$ should be mapped on $P_2 \vee P_3$. If P_2 is selected, in this case we have two internal communications. If P_3 is selected, we have one internal communication $P_1(t_1^{(1)}) \rightarrow P_1(t_2^{(1)})$ and an inter-processor communication $P_2(t_1^{(2)}) \rightarrow P_3(t_2^{(2)})$.
2) - The first replica $t_2^{(1)}$ is mapped on P_3, then both P_3 and $P_1 \vee P_2$ are locked by equation (7). Suppose that P_1 is locked, then second replica $t_2^{(2)}$ should be mapped on P_2. So we have an internal communication between $P_2(t_1^{(2)}) \rightarrow P_2(t_2^{(1)})$ and an inter-processor communication $P_1(t_1^{(1)}) \rightarrow P_3(t_2^{(2)})$.

In both scenarios, we resist to 1 failure.

All processors in $\mathbb{P}$ remain locked during the mapping process of the $\varepsilon + 1$ replicas of a task. They are unlocked only before the next step, i.e, before the CAFT algorithm is repeated for the next critical free task.

ii) *Space exclusion*: The *one-to-one mapping procedure* is based on an active replication scheme with space exclusion. Thus, each task is replicated θ times onto θ distinct processors. We have at most $\theta - 1$ processor failures at the same time. So at least one copy of each task is executed on a fault free processor. □

Theorem 5.1 *The time complexity of CAFT is:*

$$O\Big(em(\varepsilon+1)^2 \log(\varepsilon+1) + v \log \omega\Big)$$

Proof: The proof is composed of two parts:

i) *One-To-One Mapping Procedure (Algorithm 5.2)*: The main computational cost of this procedure is spent in the while loop (Lines 2 to 7). Line 5 costs $O(|\Gamma^-(t_i)|m)$, since all the instances/replicas in T of the immediate predecessors t_j of task t_i need to be examined

on each processor $P_k, k = 1 \dots m$. Line 3 costs $O\big(|\Gamma^-(t_i)|(\varepsilon+1)\log(\varepsilon+1)\big)$ for sorting the lists $\overline{\mathcal{B}}(t_j)$, $1 \leq j \leq |\Gamma^-(t_i)|$. Line 7 costs $O\big(|\Gamma^-(t_i)|\log(\varepsilon+1)\big)$ for finding the head of the lists $\overline{\mathcal{B}}(t_j)$, $1 \leq j \leq |\Gamma^-(t_i)|$. Thus, the cost of this procedure for the whole m loops is $O\big(|\Gamma^-(t_i)|(\varepsilon+1)m\log(\varepsilon+1)\big)$.

ii) *CAFT (Algorithm 5.1)*: Computing $b\ell(t)$ (line 4) takes $O(e+v)$. Insertion or deletion from α costs $O(\log|\alpha|)$ where $|\alpha| \leq \omega$, the width of the task graph, i.e., the maximum number of tasks that are independent in G. 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 CAFT, the time complexity for α management is in $O(v\log\omega)$. The main computational cost of CAFT is spent in the while loop (Lines 8 to 24). This loop is executed v times. Line 9 costs $O(\log\omega)$ for finding the head of α. Line 10 costs $|\Gamma^-(t_i)|m$ to determine λ_j. The two loops (12 to 15) and (16 to 20) are excuted $\varepsilon+1$ times. Line 17 costs $O(|\Gamma^-(t_i)|(\varepsilon+1)m)$, since all the instances/replicas of the immediate predecessors t_j of task t_i need to be examined on each processor $P_k, k = 1 \dots m$. Line 21 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 CAFT for the v tasks in G is

$$\begin{aligned}\sum_{i=1}^{v} O(|\Gamma^-(t_i)|m(\varepsilon+1)^2\log(\varepsilon+1) + e + v\log\omega)\\ = O\Big(em(\varepsilon+1)^2\log(\varepsilon+1) + v\log\omega\Big)\end{aligned}$$

Because $\varepsilon < m$, we derive the upper bound:

$$O\Big(em^3\log m + v\log\omega\Big)$$

□

6 Experimental results

We assess the practical significance and usefulness of the CAFT algorithm through simulation studies. We compare the performance of CAFT with the two most relevant fault tolerant scheduling algorithms, namely FTSA and FTBAR. We use randomly generated graphs, whose parameters are consistent with those used in the literature [10, 21]. We characterize these random graphs with three parameters: (i) the number of tasks, chosen uniformly from the range $[80, 120]$; (ii) the number of incoming/outgoing edges per task, which is set in $[1, 3]$; and (iii) the granularity of the task graph $g(G)$. The granularity indicates the ratio of the average computation time of the tasks to that of communication time. We consider two types of graphs, with a granularity (A) in $[0.2, 2.0]$ and increments of 0.2, and (B) in $[1, 10]$ and increments of 1. Two types of platforms are considered, first with 10 processors and $\varepsilon = 1$ or $\varepsilon = 3$, and then with 20 processors and $\varepsilon = 5$. 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. The fault free schedule is defined as the schedule generated by each algorithm without replication, assuming that the system is completely safe. For each algorithm, we compare the fault free version (without replication) and the fault tolerant algorithm. Note that the fault-free version of CAFT reduces to an implementation of HEFT, the reference heuristic in the literature [27]. Also recall that the upper bounds of the schedules are computed as explained in Section 4.2 or [4]. Each algorithm is evaluated in terms of achieved latency and fault

tolerance overhead, given by the following formula:
Overhead =

$$\frac{\text{CAFT}^0|\text{FTSA}^0|\text{FTBAR}^0|\text{CAFT}^c|\text{FTSA}^c|\text{FTBAR}^c - \text{CAFT}^*}{\text{CAFT}^*}$$

where the superscripts $*$, c and 0 respectively denote the latency achieved by the fault free schedule, the latency achieved by the schedule when processors effectively fail (crash) and the latency achieved with 0 crash.

Note that for each algorithm, 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.

Figures 1(a), 2(a) and 3(a) clearly show that CAFT outperforms both FTSA and FTBAR. These results indicate that network contention has a significant impact on the latency achieved by FTSA and FTBAR. This is because 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 and FTBAR, a quadratic increase. In contrast, the CAFT algorithm is not really sensitive to the contention since it uses the one-to-one mapping procedure to reduce this overhead down to, in the most favorable cases, a linear number $e(\varepsilon+1)$ of communications. In addition, we find that CAFT achieves a really good latency (with 0 crash), which is quite close to the fault free version. As expected, its upper bound is close to the latency with 0 crash since we keep only the best communication edges in the schedule.

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 (upper bound and latency with 0 crash). Processors that fail during the schedule process are chosen uniformly from the range $[1, 10]$. The first observation from Figures 1(b) and 2(b) is that even when crash occurs, CAFT^c behaves always better than FTSA^c and FTBAR^c. This is because CAFT accounts for communication overhead during the mapping process by removing some of the communications. The second interesting observation is that the latency achieved by both FTSA and FTBAR compared to the schedule length generated with 0 crash sometimes increases (see 1(b)) and other times decreases (2(b)). To explain this phenomenon, consider the example of a simple task graph composed of three tasks in precedence $(t_1 \prec t_3) \wedge (t_2 \prec t_3)$ and $P = \{P_m, 1 \leq m \leq 6\}$. Assume that $\varepsilon = 1$, $\mathcal{B}(t_1) = \{t_1^{(1)}, t_1^{(2)}\}$, $\mathcal{B}(t_2) = \{t_2^{(1)}, t_2^{(2)}\}$ and $\mathcal{B}(t_3) = \{t_3^{(1)}, t_3^{(2)}\}$ with $\mathcal{P}\big(\mathcal{B}(t_1)\big) = \{P_1, P_2\}$, $\mathcal{P}\big(\mathcal{B}(t_2)\big) = \{P_3, P_4\}$ and $\mathcal{P}\big(\mathcal{B}(t_3)\big) = \{P_5, P_6\}$ respectively. Assume that the latency achieved with 0 crash is determined by the replica of $t_3^{(1)}$.

For the sake of simplicity, assume that the sorted list of the instances/replicas of both $\mathcal{B}(t_1)$ and $\mathcal{B}(t_2)$ (sorting is done by non decreasing order of their communication finish time $\mathcal{F}(c, l)$ on the links) are in this order $\{t_1^{(1)}, t_1^{(2)}, t_2^{(1)}, t_2^{(2)}\}$. $t_3^{(1)}$ will receive its input data 4 times. But as soon as it receives its input data from $t_2^{(1)}$, the task is executed and ignores the later incoming data from $t_2^{(2)}$. So, without failures and since communications are serialized at the reception, 3 communications are taken into account so that $t_3^{(1)}$ can run earlier. But, in the presence of 1 failure, two scenarios are possible:
i) if P_2 fails, the finish time of the replica $t_3^{(1)}$ will be sooner than its estimated finish time; ii) if P_2 and P_3 fail, the start time of the replica is delayed until the arrival of its input data from $t_2^{(2)}$. This leads to an increase of its finish time and consequently to an increase of the latency achieved with crash.

Applying this reasoning to all tasks of G, the impact of processors crash are spread throughout the execution of the application, which may lead either to a reduction or to an increase of the schedule length.

This behavior is identical when we consider larger platforms, as illustrated in Figure 3.

We also evaluated the impact of the granularity on performance of each algorithm. Thus, Figures 4, 5 and 6 reveal that when the $g(G)$ value is small, the latency of CAFT is significantly better than that of FTSA and FTBAR. This is explained by the fact that for small $g(G)$ values, i.e. high communication costs, contention plays quite a significant role. However, the impact of contention becomes less important as the granularity $g(G)$ increases, since larger $g(G)$ values result in smaller communication times. Consequently, the fault tolerance overhead of FTSA diminishes gradually and becomes closer to that of CAFT as the $g(G)$ value goes up. However, the fault tolerance overhead of FTBAR increases with the increasing values of the granularity. 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.

Finally, we readily observe from all figures 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. In addition, it is interesting to note that when the number of failures increases, there is not really much difference in the increase of the latency achieved by CAFT, compared to the schedule length generated with 0 crash. 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.

To summarize, the simulation results show that CAFT is considerably superior to the other two algorithms in all the cases tested ($0.2 \leq g(G) \leq 10, m = \{10, 20\}$). They also indicate that network contention has a significant impact on the latency achieved by FTSA and FTBAR. Thus, this experimental study validates the usefulness of our algorithm CAFT, and confirms that when dealing with realistic model platforms, contention should absolutely be considered in order to obtain improved schedules. To the best of our knowledge, the proposed algorithm is the first to address both problems of network contention and fault-tolerance scheduling.

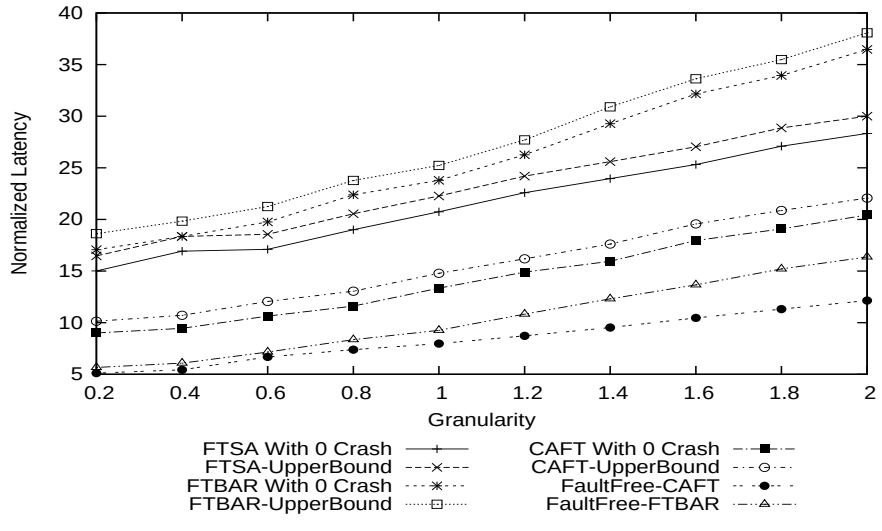
7 Conclusion

In this paper we have presented CAFT, an efficient fault-tolerant scheduling algorithm for heterogeneous systems based on an active replication scheme. CAFT is able to dramatically reduce the communication overhead induced by task replication, which turns out a key factor in improving performance when dealing with realistic, communication contention aware, platform models.

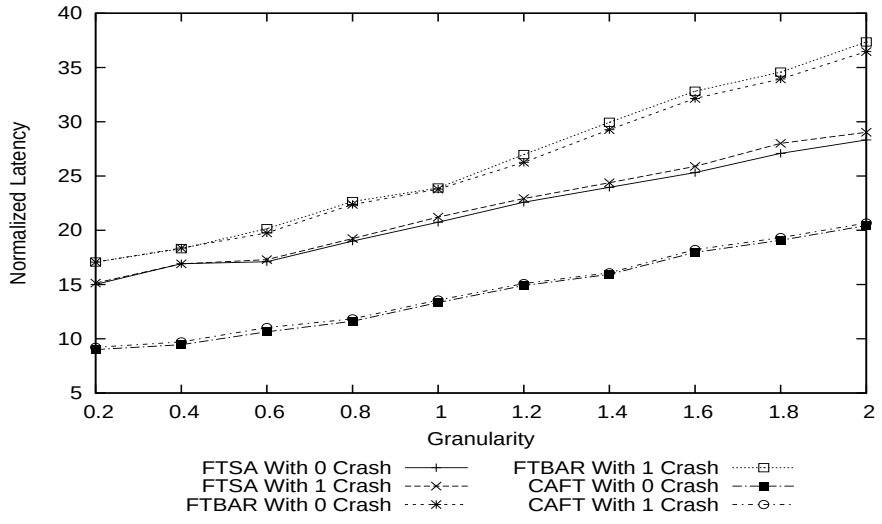
To assess the performance of CAFT, simulation studies were conducted to compare it with (the one-port adaptation of) FTBAR and FTSA, which seem to be its main direct competitors from the literature. We have shown that CAFT is very efficient both in terms of computational complexity and quality of the resulting schedule.

An easy extension of CAFT would be to adapt it to sparse interconnection graphs (while we had a clique in this paper). On such platforms, each processor is provided with a routing table which indicates the route to be used to communicate with another processor. To achieve contention awareness, at most one message can circulate on a given link at a given time-step, so we need to schedule long-distance communications carefully.

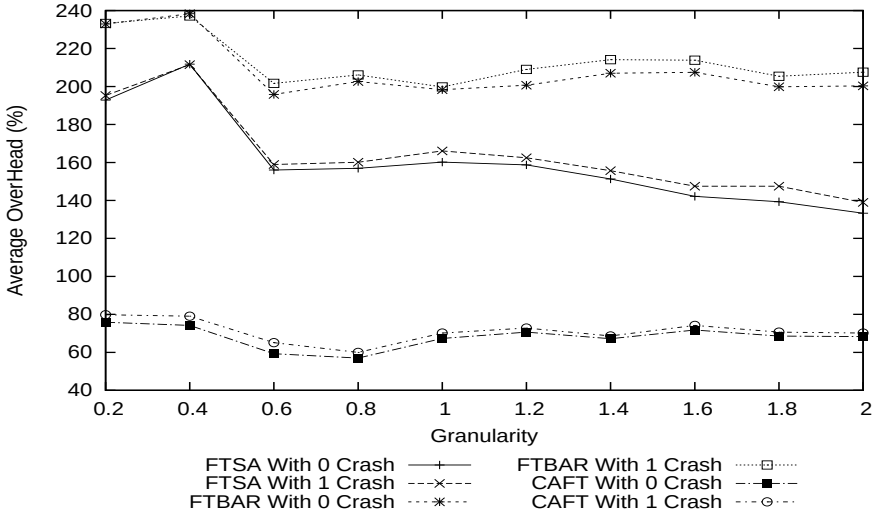
Further work will be devoted to implementing more complex heuristics that depart from the main principle of list scheduling heuristics. Instead of considering a single task (the one with highest priority) and assigning all its replicas to the currently best available resources, why not consider say, 10 ready tasks, and assign all their replicas in the same decision making procedure? The idea would be to design an extension of the one-to-one mapping procedure to a set of independent tasks, in order to better load balance processor and link usage.



(a)

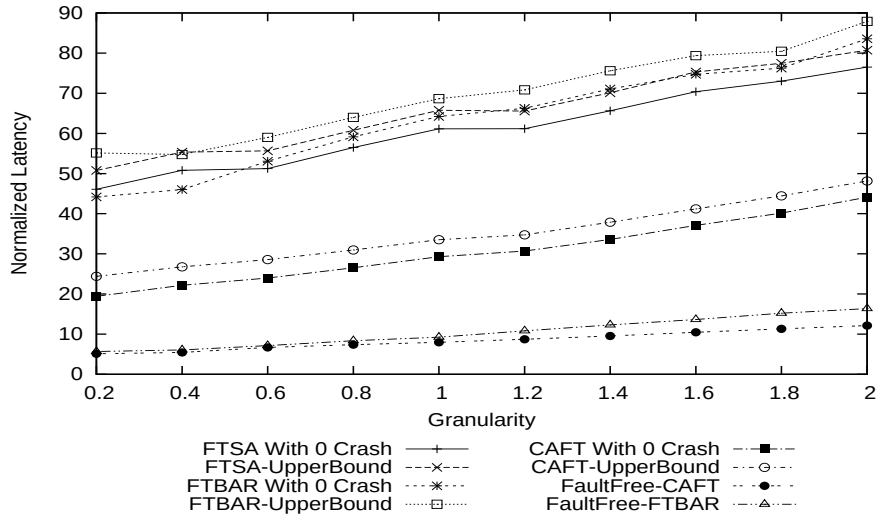


(b)

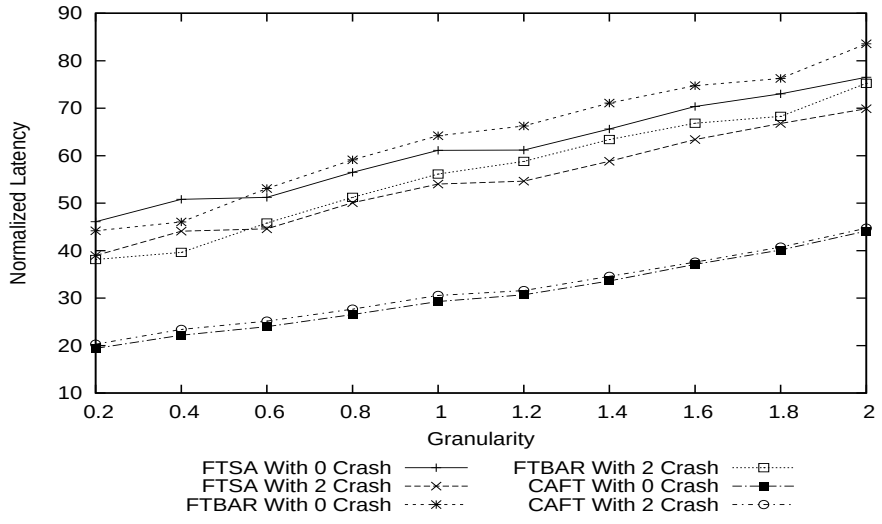


(c)

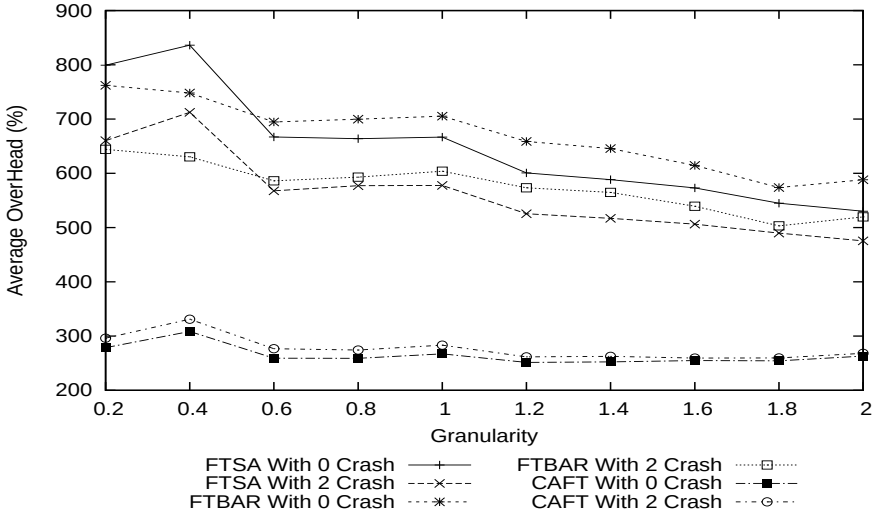
Figure 1: Average normalized latency and overhead comparison between CAFT, FTSA and FTBAR (Bound and Crash cases, $\varepsilon = 1$)



(a)

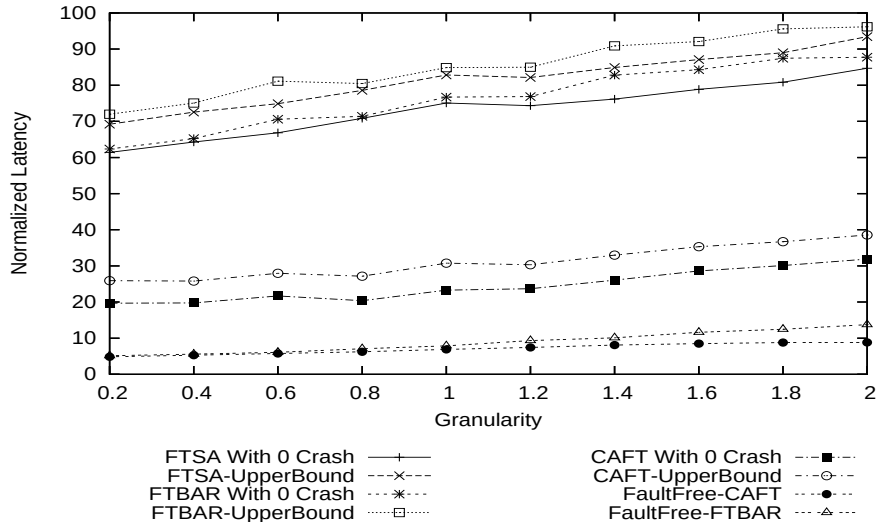


(b)

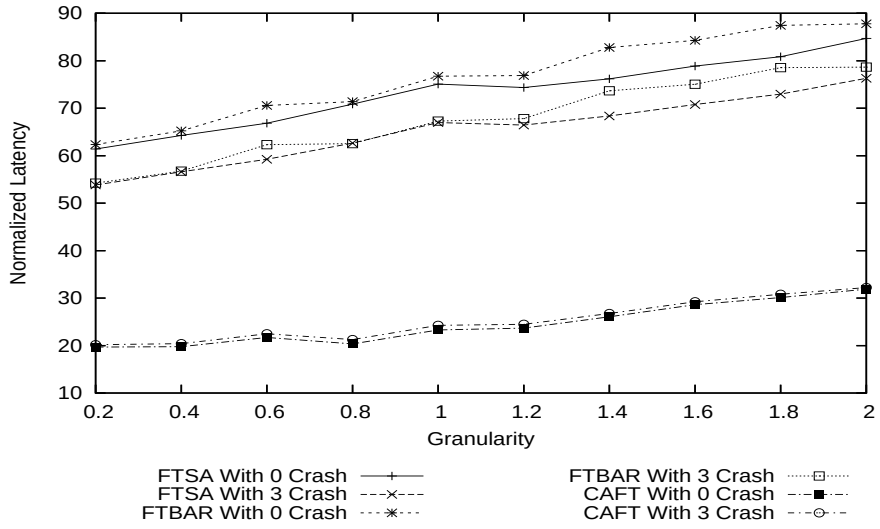


(c)

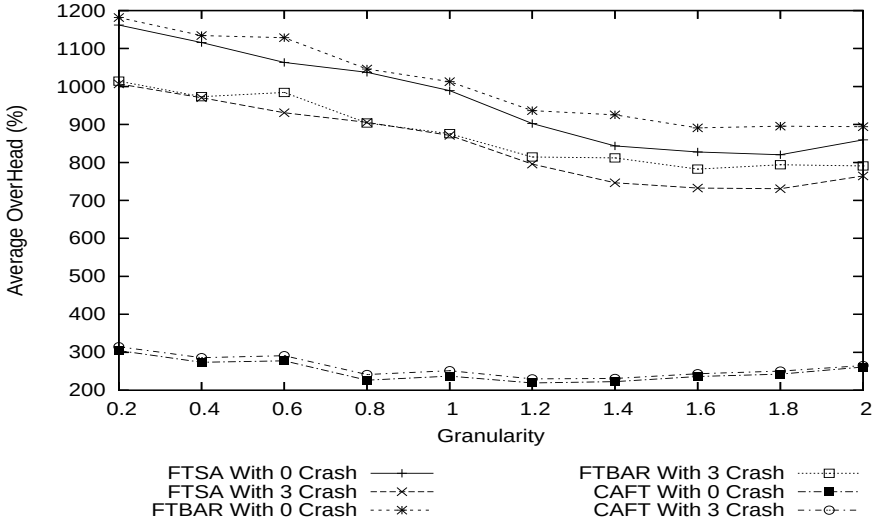
Figure 2: Average normalized latency and overhead comparison between CAFT, FTSA and FTBAR (Bound and Crash cases, $\varepsilon = 3$)



(a)

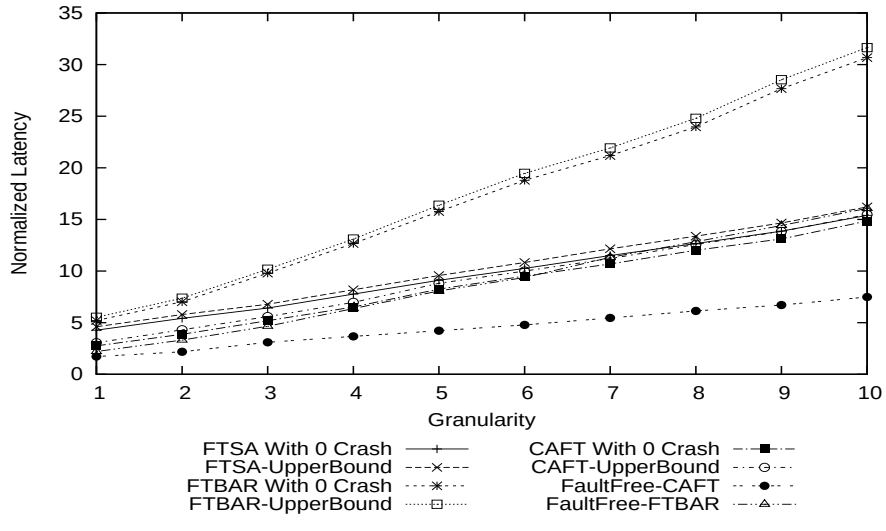


(b)

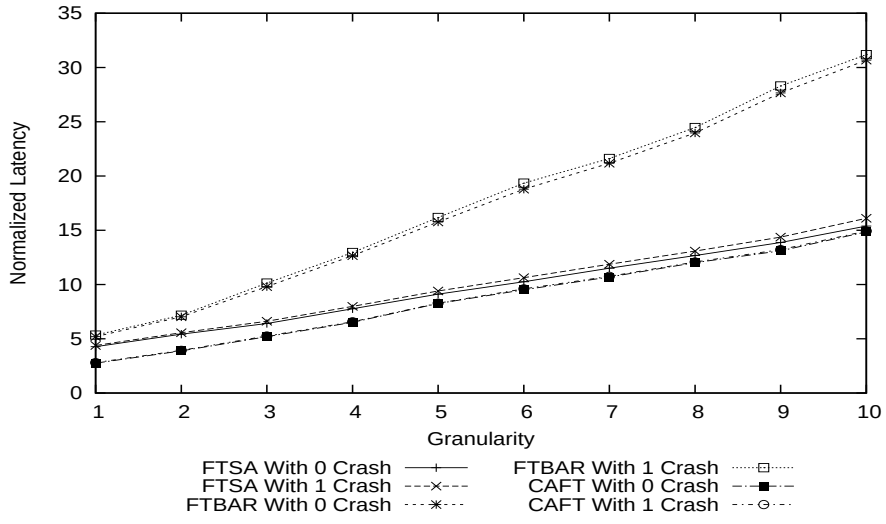


(c)

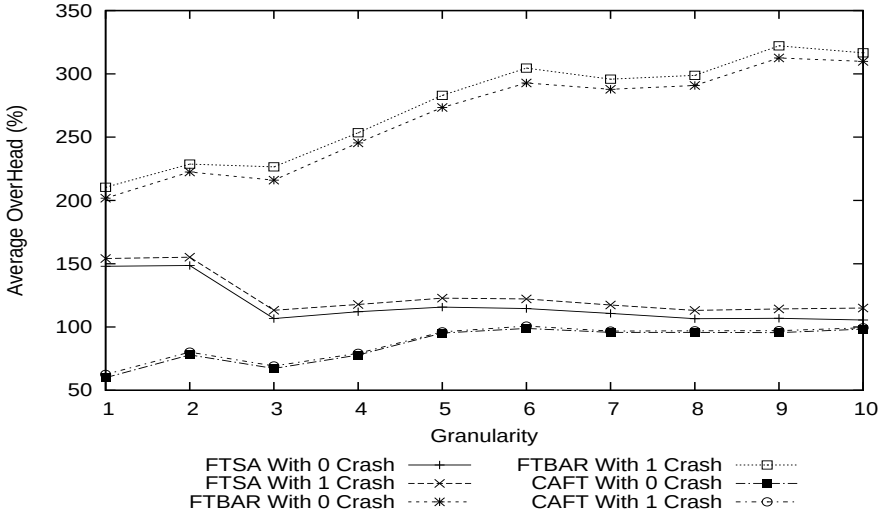
Figure 3: Average normalized latency and overhead comparison between CAFT, FTSA and FTBAR (Bound and Crash cases, $\varepsilon = 5, m = 20$)



(a)

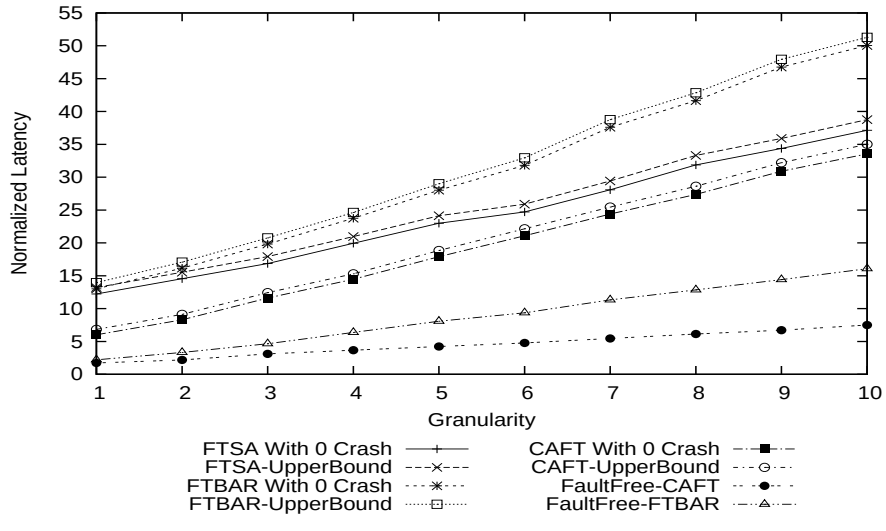


(b)

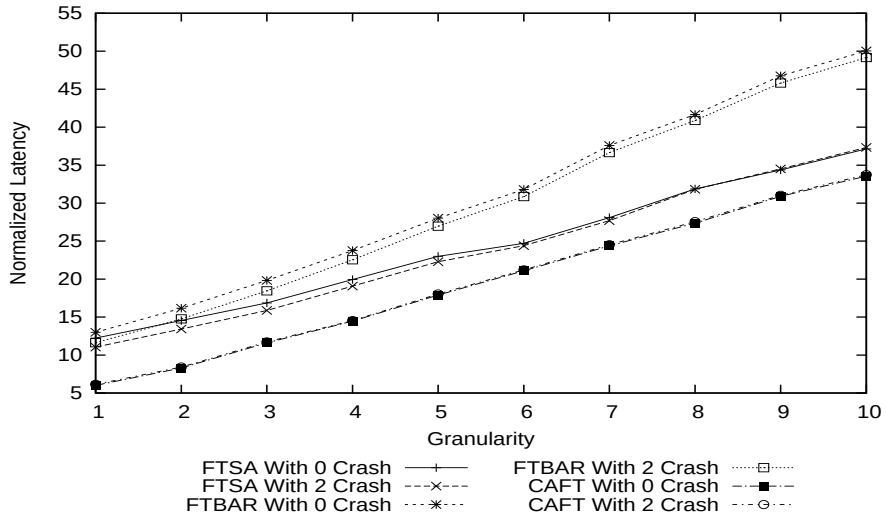


(c)

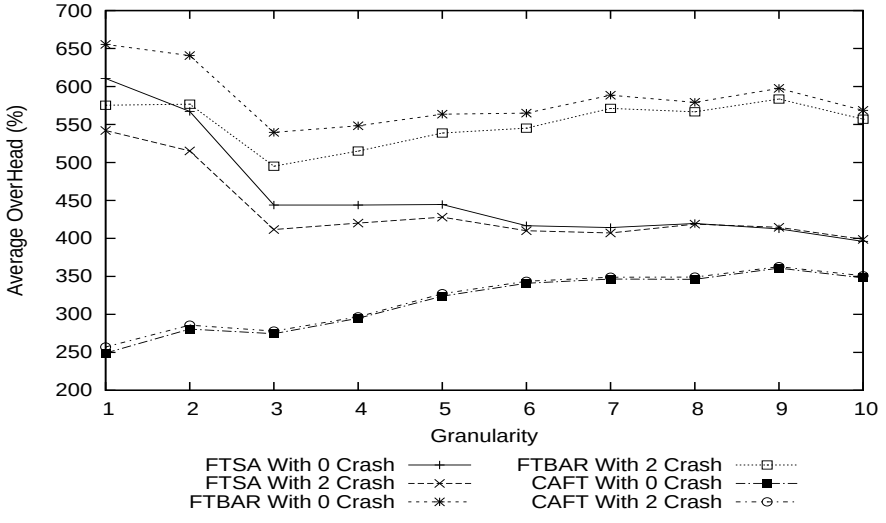
Figure 4: Average normalized latency and overhead comparison between CAFT, FTSA and FTBAR (Bound and Crash cases, $\varepsilon = 1$)



(a)

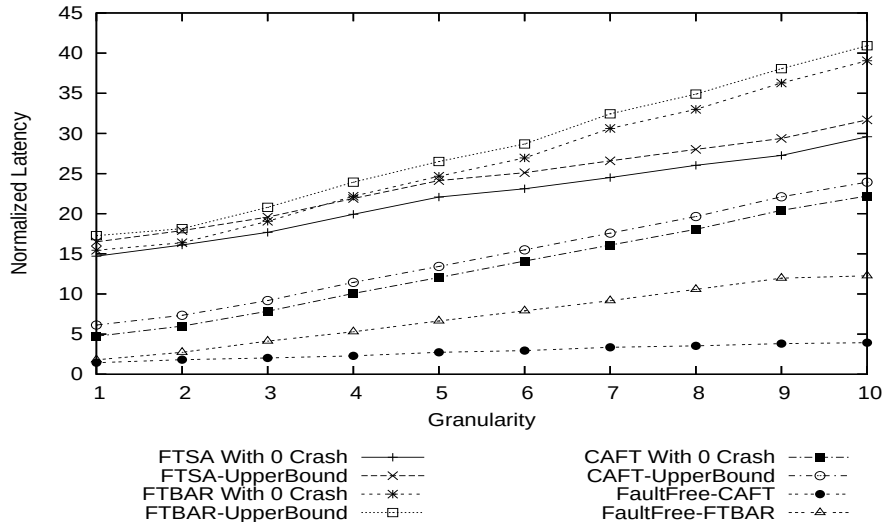


(b)

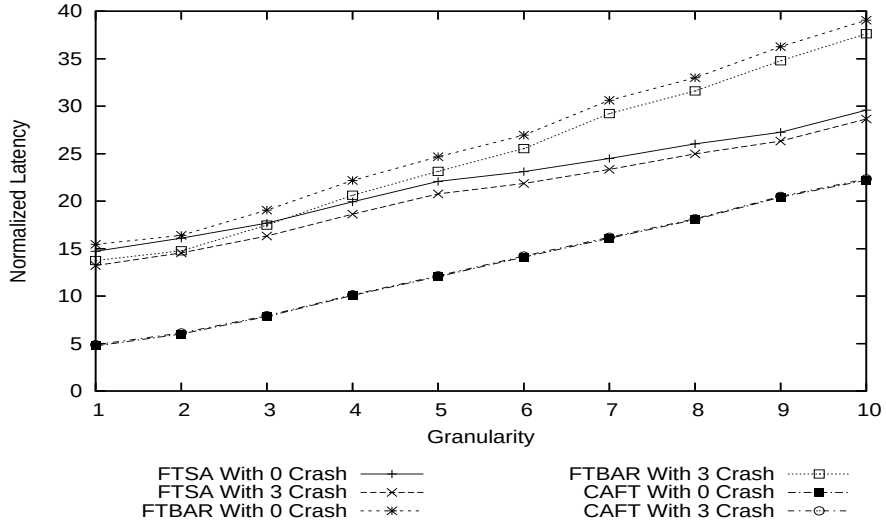


(c)

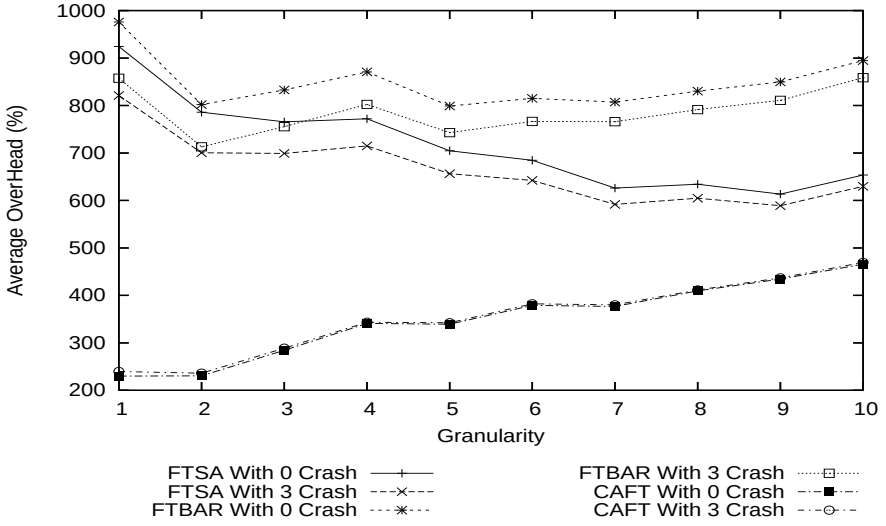
Figure 5: Average normalized latency and overhead comparison between CAFT, FTSA and FTBAR (Bound and Crash cases, $\varepsilon = 3$)



(a)



(b)



(c)

Figure 6: Average normalized latency and overhead comparison between CAFT, FTSA and FTBAR (Bound and Crash cases, $\varepsilon = 5, m = 20$)

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] M. Banikazemi, V. Moorthy, and D. K. Panda. Efficient collective communication on heterogeneous networks of workstations. In *Proceedings of the 27th International Conference on Parallel Processing (ICPP'98)*. IEEE Computer Society Press, 1998.

[4] Anne Benoit, Mourad Hakem, and Yves Robert. Fault Tolerant Scheduling of Precedence Task Graphs on Heterogeneous Platforms. Research Report 2008-03, LIP, ENS Lyon, France, January 2008. Available at graal.ens-lyon.fr/~abenoit/.

[5] P.B. Bhat, C.S. Raghavendra, and V.K. Prasanna. Efficient collective communication in distributed heterogeneous systems. In *ICDCS'99 19th International Conference on Distributed Computing Systems*, pages 15–24. IEEE Computer Society Press, 1999.

[6] P.B. Bhat, C.S. Raghavendra, and V.K. Prasanna. Efficient collective communication in distributed heterogeneous systems. *Journal of Parallel and Distributed Computing*, 63:251–263, 2003.

[7] P. Chrétienne, E. G. Coffman Jr., J. K. Lenstra, and Z. Liu, editors. *Scheduling Theory and its Applications*. John Wiley and Sons, 1995.

[8] H. El-Rewini, H. H. Ali, and T. G. Lewis. Task scheduling in multiprocessing systems. *Computer*, 28(12):27–37, 1995.

[9] 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.

[10] 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.

[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] L. Hollermann, T. S. Hsu, D. R. Lopez, and K. Vertanen. Scheduling problems in a practical allocation model. *J. Combinatorial Optimization*, 1(2):129–149, 1997.

[14] 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.

[15] T. S. Hsu, J. C. Lee, D. R. Lopez, and W. A. Royce. Task allocation on a network of processors. *IEEE Trans. Computers*, 49(12):1339–1353, 2000.

[16] S. Khuller and Y.A. Kim. On broadcasting in heterogenous networks. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1011–1020. Society for Industrial and Applied Mathematics, 2004.

[17] P. Liu. Broadcast scheduling optimization for heterogeneous cluster systems. *Journal of Algorithms*, 42(1):135–152, 2002.

[18] 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.

[19] 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.

[20] M. G. Norman and P. Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):103–117, 1993.

[21] 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.

[22] T. Saif and M. Parashar. Understanding the behavior and performance of non-blocking communications in MPI. In *Proceedings of Euro-Par 2004: Parallel Processing*, LNCS 3149, pages 173–182. Springer, 2004.

[23] B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Science Press, 1995.

[24] O. Sinnen and L. Sousa. Experimental evaluation of task scheduling accuracy: Implications for the scheduling model. *IEICE Transactions on Information and Systems*, E86-D(9):1620–1627, 2003.

[25] O. Sinnen and L. Sousa. Communication contention in task scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):503–515, 2005.

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

[27] H. Topcuoglu, S. Hariri, and M. Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distributed Systems*, 13(3):260–274, 2002.

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