HAL
open science

Optimizing the Latency of Streaming Applications under Throughput and Reliability Constraints

Anne Benoit, Mourad Hakem, Yves Robert

▶ To cite this version:

Anne Benoit, Mourad Hakem, Yves Robert. Optimizing the Latency of Streaming Applications under Throughput and Reliability Constraints. 2009. ensl-00376968

HAL Id: ensl-00376968
https://ens-lyon.hal.science/ensl-00376968

Preprint submitted on 20 Apr 2009

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.

Optimizing the Latency of Streaming Applications under Throughput and Reliability Constraints

Anne Benoit[1], Mourad Hakem[2] and Yves Robert[1]
[1] LIP laboratory, ENS Lyon – CNRS – INRIA – UCBL, Lyon, France
[2] LIFC Laboratory, Université de Franche-Comté, Belfort, France
{Anne.Benoit,Mourad.Hakem,Yves.Robert}@ens-lyon.fr

April 20, 2009

LIP Research Report RR-2009-13

Abstract

In this paper, we deal with the problem of scheduling streaming applications on unreliable heterogeneous platforms. We use the realistic one-port model with full computation/communication overlap. We deal with three optimization objectives. The first two, latency and throughput, are performance-related while the third, tolerating a given number of processor failures, is reliability-oriented. The major contribution of this paper is the design of a new scheduling algorithm to minimize latency under both throughput and reliability constraints. We provide a comprehensive set of experimental results, that fully demonstrate the usefulness of the proposed algorithm.

1 Introduction

Pipelined workflows are a popular programming paradigm for streaming applications like video and audio encoding and decoding, DSP applications, etc [11, 5]. Streaming applications are becoming increasingly prevalent, and many languages are being continually designed to support these applications. In these languages, the programmer expresses programs by creating a *workflow graph*, and the system maps this workflow graph on a target machine. A workflow graph contains several *tasks*, and these tasks are connected to each other using first-in-first-out *channels*. Data sets are input into the graph using input channel(s) and the outputs are produced on the output channel(s). Since data continually flows through these streaming applications, the goal of a scheduler is often to decrease the *latency* and/or increase the *throughput*. Here the latency, or response time, is defined as the time for a single data item to traverse the graph, that is, to execute all the tasks of the application. Latency is typically important for the end-user who is waiting for the results. The throughput is the aggregate rate at which the input data stream is processed. The inverse of the throughput is the period, defined as the time-interval between two consecutive data sets entering the system. Achieving a high throughput is a typical requirement for real-time applications and usually leads to an efficient utilization of hardware resources.

Latency and throughput are the main performance-related scheduling objectives, and they are conflicting criteria. Indeed, in the absence of throughput constraints, the latency is the longest path in the execution graph: then an optimal strategy for latency minimization is to map the whole graph onto the fastest processor, thereby eliminating all communications and reducing the computing cost as much as possible. But then the period is equal to the latency, and the throughput may well become dramatically low. Real-life problems often call for bi-criteria optimization problems, such as minimizing the latency while enforcing a minimum throughput. With the advent of large-scale heterogeneous platforms, another important objective is to achieve a reliable execution. This objective is not related to performance, contrarily to latency/throughput optimization. Instead, the goal is tolerate a given number of processor failures. Our approach is based on an active replication scheme, capable of supporting ε arbitrary fail-silent (a faulty processor does not produce any output) and fail-stop (no processor recovery) processor failures.

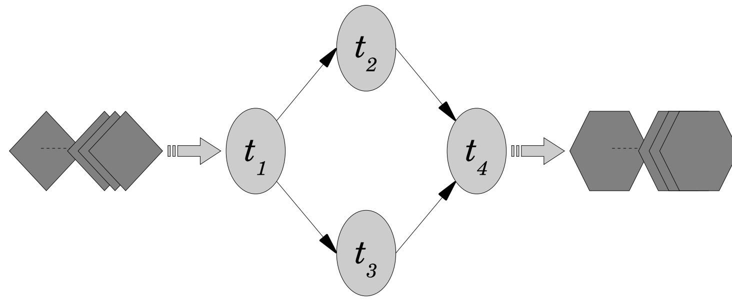
Here is an example to illustrate several execution scenarios, and to outline the differences between task and data parallelism for an application graph, and pipelined execution of successive instances of the same graph. The workflow is shown in Fig. 1(a). All task computation times are equal to 15, and all edges have a communication volume equal to 2. We have four processors P_1 to P_4 whose speeds are $s_1 = s_3 = 1.5$ and $s_2 = s_4 = 1$. All links have unit bandwidth. The fault tolerance degree is $\varepsilon = 1$, so that each task is replicated once: $t_i^{(1)}$ represents the first copy of task t_i, while $t_i^{(2)}$ is the second copy, which is always executed but turns out useful only if a failure occurs.

i) Task parallelism– To minimize the makespan of the DAG graph, we use classical list scheduling techniques [9], leading to the assignment of Fig. 1(b). In streaming mode, repeating the execution for incoming data sets, we obtain a latency $\mathcal{L} = 39$ and a throughput $\mathcal{T} = 1/39$.

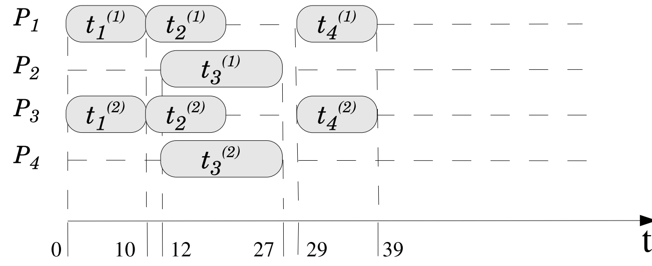
ii) Data parallelism– All tasks in the DAG are mapped to a single processor, we make four replicas, and consecutive instances of the input stream are distributed to the processors in round-robin fashion (Fig. 1(c)). In the absence of failures, the maximum throughput is $\mathcal{T} = 2/40 = 1/20$. However, this technique requires that the processing of one data item is independent of the results obtained for the previous data item, a drastic assumption that we do not make.

iii) Pipelined execution– Fig. 1(d) shows a mapping with $\mathcal{S} = 2$ synchronous stages (t_1, t_3) and (t_2, t_4) which are executed in parallel once the pipeline is filled. The throughput is $\mathcal{T} = 1/30$ and the latency is $\mathcal{L} = \frac{2\mathcal{S}-1}{\mathcal{T}} = 90$ (see Section 4 for an explanation of this value). The advantage of this technique is that it can be applied to either dependent or independent data items. It is the one used in the literature for streaming applications (see the related work in Section 3), and we use it in the following too.

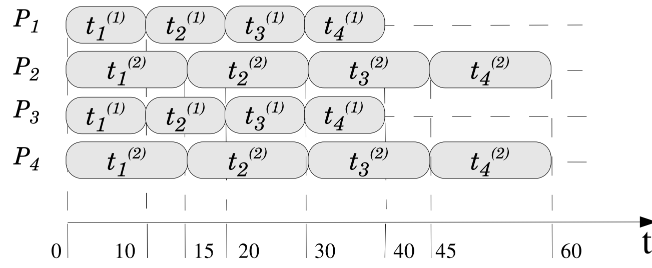
After some definitions and notations in Section 2, we present in Section 3 a brief survey of heuristics proposed in the literature to optimize latency under throughput constraints. These heuristics target homogeneous platforms and assume unlimited network capacity. Instead, we suggest to use a realistic communication model, the bi-directional one-port model with full computation/communication overlap. In addition, we introduce a third, reliability-oriented, objective, that of tolerating a given number ε of processor failures. The major contribution of this paper is the design of a new scheduling algorithm to minimize latency under both throughput and reliability constraints (Section 4). We provide in Section 5 a comprehensive set of experimental results, that fully demonstrate the usefulness of the proposed algorithm. Finally we give concluding remarks in Section 6.



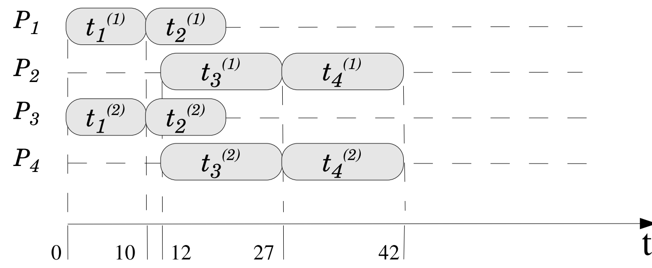
(a) - Workflow graph



(b) - Task parallelism



(c) - Data parallelism



(d) - Pipelined execution

Figure 1: Different Mappings.

2 Framework

The application graph is a weighted Directed Acyclic Graph (DAG) $G = (V, E)$, where V is the set of nodes, or tasks, and E is the set of edges corresponding to precedence relations between tasks; $v = |V|$ 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, $\mathcal{E}(t_i)$ is its execution time, $\Gamma^-(t)$ is the set of its immediate predecessors and $\Gamma^+(t)$ the set of its immediate successors. A task is called *ready* if it is unscheduled and all of its predecessors are scheduled. We target a heterogeneous platform with m processors $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$, fully interconnected. The speed of P_i is s_i. The link between processors P_k and P_h is denoted by l_{kh} and has bandwidth d_{kh}. Note that we do not need physical links between processor pairs, we may have a switch, or even a path composed of several physical links to interconnect P_k and P_h; in the latter case we would retain the bandwidth of the slowest link in the path for the bandwidth of l_{kh}. We use the bi-directional one-port architectural model [2], 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.

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. $\mathcal{H}(\ell)$ 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 number of tasks that can be simultaneously ready 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 $|\ell| \leq \omega$. The mapping matrix $\mathcal{X}$ is a $v \times m$ binary matrix representing the mapping of the v tasks of G to the m processors. Element $\mathcal{X}_{iu}$ is equal to 1 if a copy of task t_i has been mapped to processor P_u, and 0 otherwise.

Task priorities are 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 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. The bottom level is the length of the longest path starting at task t to an exit node in the graph. The bottom level of an exit node is equal to its execution time. Path lengths are defined as the average sum of edge weights and node weights [9].

The scheduling algorithms are designed to tolerate an arbitrary, but given, number ε of processor failures. Our approach is based on an active replication scheme, where each task is replicated ε times, and executed $\varepsilon + 1$ times. We enforce the rule that valid results will be provided even if ε processors fail, which calls for replicating communications as well as tasks. But communicating between any task replica pair is often useless, and minimizing communication overhead while guaranteeing valid results is a key objective of the mapping procedures described in Section 4.

3 Related work

As stated above, the following heuristics from the literature all target homogeneous platforms. This greatly simplifies all estimations of computing times and path lengths. In addition, they do not limit the number of simultaneous communications that a processor can be involved in, which also simplifies the mapping process. Still, these heuristics are insightful for our framework, namely heterogeneous platforms under the realistic one-port model.

The algorithm in [4] aims at satisfying a prescribed throughput requirement by minimizing inter-processor communications when assigning tasks to processors. It is based on the pre-clustering method similar to that in [7]. Communication edges are sorted by data volume and dealt with greedily. At each step, the algorithm attempts to match the processor executing the edge source and the processor executing the edge sink. Remaining unassigned tasks are assigned to clusters on a first-fit basis. The pre-clustering phase is followed by two refinement phases to reduce communication overhead.

The EXPERT algorithm [3] considers all paths in the application graph, and sorts them by execution time. Paths are then processed greedily. At each step, the algorithm searches for sub-paths whose tasks fit within one period, and groups these tasks into stages. Clusters are then built, first intra-stages, and then across stages, with the goal of load-balancing computations along the paths.

The TDA algorithm [11] is designed to tackle both resource and throughput optimization. A schedule is constructed to achieve the desired throughput with the minimum number of processors. A combination of two heuristics is used to solve this problem. First, the ETF (Earliest Task First) heuristic [6] is used to assign tasks to processors. Then,

a top-down approach is used to partition tasks into stages, where as before a stage is defined as a subset of tasks whose combined execution does not exceed the period. Several refinement steps are performed to improve processor utilization.

The STDP Algorithm [8] starts with one top-down and one bottom-up graph traversals to compute earliest and latest execution times for each task. Task clusters are then built with the goal of minimizing communication overhead. If some resources are still available at that point, critical tasks are then duplicated in order to decrease the latency. Finally, stages are generated through a third traversal of the graph.

The WMSH Algorithm [10] uses a clustering procedure as its first step, under the assumption that there is an unlimited number of fully interconnected processors. Then clusters are merged and scheduled on the available physical resources. In the first phase of the process, a schedule that meets the throughput requirement is obtained, assuming an unbounded number of processors. The second phase uses a processor-reduction heuristic. The third phase refines the mapping to optimize the latency, by minimizing the communication overhead along the critical path of the workflow. WSMH performs explicit task duplication to increase the throughput, while aiming at keeping communication overhead reasonably low.

The algorithm in [5] performs a binary search to find the minimal period, given the number of available processors. The search repetitively calls a mapping routine that determines how many processors are needed to execute the task graph, given the current period. This routine performs a top-down traversal, partitioning the graph into stages.

4 Scheduling Algorithms

We need a few definitions. The *processor utilization* $U_P \leq 1$ is defined as the fraction of time each processor is active. Formally, $U_{P_u} = \frac{\mathcal{T} \sum_{1 \leq i \leq v} \mathcal{X}_{iu} \mathcal{E}(t_i)}{s_u}$ for $1 \leq u \leq m$ (where $\mathcal{T}$ is the throughput). The *link utilization* U_l is defined similarly. We denote by $\mathcal{B}(t)$ the set of $\varepsilon + 1$ replicas of a task t. Also, we denote by $t^{(\mathcal{N})}$ those replicas, for $1 \leq \mathcal{N} \leq \varepsilon + 1$. Thus, $\mathcal{B}(t) = \{t^{(1)}, ..., t^{(\varepsilon+1)}\}$. $P(t^{(\mathcal{N})})$ is the processor on which replica $t^{(\mathcal{N})}$ is scheduled. For a current task t, a processor P is called *singleton* if it has only one instance/replica $t_i^{(\mathcal{N})}$, $1 \leq i \leq |\Gamma^-(t)|$, $1 \leq \mathcal{N} \leq \varepsilon + 1$; P is said *locked* either if it is already involved in a communication with a replica of t, or it processes itself one of these replicas. During the mapping steps, $X \subseteq \bigcup_{j=1}^{|\Gamma^-(t)|} \{P(\mathcal{B}(t_j))\}$ is the subset of singleton processors and $\mathbb{P} \subseteq P$ the subset of locked processors.

Informally, with $\varepsilon + 1$ replicas of each task, we could need up to $(\varepsilon + 1)^2$ communications for each edge in E, hence a total of $(\varepsilon + 1)^2 e$ communications. To reduce this number, we use a strategy similar to [1]: while there are enough singleton processors with replicas of predecessor tasks, we use the one-to-one mapping procedure described in Algorithm 4.2. This name stems from the fact that each replica in $\bigcup_{i=1}^{|\Gamma^-(t)|} \overline{\mathcal{B}}(t_i)$ should communicate to exactly one replica in $\mathcal{B}(t)$. The number of times θ that the one-to-one-mapping procedure is called for scheduling the $\varepsilon + 1$ replicas of the current task is given as $\theta \leftarrow \min_i(\lambda_i)$, where $\overline{\mathcal{B}}(t_i)$ is the subset of replicas of each predecessor t_i scheduled in $\mathcal{X}$ and λ_i its cardinality $\left(\lambda_i = |\overline{\mathcal{B}}(t_i)|\right)$.

The inverse of the throughput is the iteration period Δ, which corresponds to the time-interval between the processing of two consecutive data items. Formally, the cycle-time of processor P_u, $1 \leq u \leq m$, is defined as $\Delta_u = \max\left(\Sigma_u,\ \mathcal{C}_u^{I/O}\right)$ where Σ_u is the computing load of P_u and $\mathcal{C}_u^{I/O}$ is the input/output communication cycle time of processor P_u, $1 \leq u \leq m$. The throughput achieved under the mapping $\mathcal{X}$ is $\mathcal{T} = \frac{1}{\max_{1 \leq u \leq m} \Delta_u}$.

To compute the latency, we borrow the notion of pipeline *stages* to [4]. Intuitively, stages record processor changes along dependence paths in the application graph. The pipeline stage $\mathcal{S}^{(\mathcal{N})}$ of task/replica $t^{(\mathcal{N})}$, $1 \leq \mathcal{N} \leq \varepsilon + 1$ depends on stage of those predecessors $t_*^{(\mathcal{N})}, t_* \in \Gamma^-(t)$, involved in a communication with $t^{(\mathcal{N})}$. Entry tasks/replicas are mapped in the first stage. The stage of the other tasks/replicas is computed as $\mathcal{S}^{(\mathcal{N})} = \max\{\mathcal{S}_*^{(\mathcal{N})} + \eta\}$, where $\eta = 0$ if $P\left(t_*^{(\mathcal{N})}\right) = P\left(t^{(\mathcal{N})}\right)$ and $\eta = 1$ otherwise. Then the latency $\mathcal{L}$ depends on the total number of stages $\mathcal{S}$ and the desired throughput $\mathcal{T}$. It is given [4] by $\mathcal{L} = \frac{2\mathcal{S}-1}{\mathcal{T}}$.

In the following, we present two heuristics. The first one, LTF, aims at reducing the communication overhead while the second one, Reverse LTF, also aims at keeping the total number of stages as low as possible.

4.1 The LTF Algorithm

The LTF (Latency, Throughput, Failures) algorithm is essentially an extended version of the Iso-Level CAFT algorithm of [1], which tackles the combination of communication overhead reduction and fault tolerance requirements. It differs from the initial version in the way that it takes the throughput requirement into account. Tasks are assigned to processors not only to achieve fault tolerance and latency requirements, but also to satisfy the desired throughput performance of the application. Tasks are scheduled and partitioned into pipeline stages greedily. Algorithm 4.1 outlines the pseudocode of the LTF heuristic. The input of the algorithm is a task graph G, the fault tolerance degree ε and a desired throughput $\mathcal{T}$.

At each step of the mapping process, LTF selects a subset β of ready tasks with highest priority, and simulates the mapping of each task in the subset on all processors. Working with a subset rather than with a single task (as classical list-scheduling algorithms) allows for a better load balance [1]. For each task $t \in \beta$, we search for unlocked processors which can execute t without exceeding the desired iteration period. Formally:

$$\begin{array}{l}(\mathcal{T} \cdot \Sigma_u \leq 1) \wedge \left(\mathcal{T} \cdot \mathcal{C}_u^I \leq 1\right) \wedge \left(\mathcal{T} \cdot \mathcal{C}_h^O \leq 1\right) \wedge \left(P_u \notin \mathbb{P}^k\right) \\ 1 \leq u, h \leq m,\ P(t) = u, P(t_*) = h,\ u \neq h,\ t_* \in \Gamma^-(t)\end{array} \quad (1)$$

If there are several such processors, we select the one with minimum finish time $\mathcal{F}$. If there are none, we use other processors, at the risk of increasing the communication overhead. The algorithm fails if no processor can accommodate the task because of the throughput constraint. The time complexity of LTF Algorithm is given below:

Theorem 1 *The time complexity of LTF is* $O\left(em(\varepsilon+1)^2 \log(\varepsilon+1) + v \log \omega\right)$.

The proof is similar to that given in [1] for Iso-Level CAFT. Note that $\varepsilon < m$, and that the width ω does not exceed v, so we derive the upper bound $O\left(em^3 \log m + v \log v\right)$.

4.2 Reverse LTF Algorithm

As stated above, we have to reduce the number of stages $\mathcal{S}$ as much as possible to optimize the pipeline latency $\mathcal{L}$. This is the goal of the R-LTF algorithm (R for Reverse) that we introduce now. It consists of a sequence of refinement steps, where each step creates a new pipeline stage or grows an existing one. Unlike LTF, the R-LTF uses a bottom-up topological traversal of the application graph, starting from sink nodes. R-LTF mapping decisions are guided by two main rules, which are invoked in the order below:

- **Rule 1:** The pipeline stage number $\left(\max\limits_{t_* \in \Gamma^+(t)} \mathcal{S}\right)$ of the current task/replica t does not increase when scheduling it.
- **Rule 2:** The number of communications induced by the replication mechanism should be reduced as much as possible. If t is the current task to be scheduled and t' one of its successors, we check whether

$$\left(|\Gamma^+(t)| = 1\right) \wedge \left(\forall t_* \in \Gamma^-(t'), |\Gamma^+(t_*)| = 1, t_* \in \alpha\right)$$

If this condition holds, we assign all replicas of t with the one-to-one mapping procedure.

Note that by applying the latter rule in the absence of throughput constraints, we can reduce the number of communications down to $e(\varepsilon+1)$ for any series-parallel graph (the proof is similar to that given in [1]). Finally, note that the complexity of R-LTF is the same as that of LTF.

4.3 Example

In this section we work out an example to illustrate the difference between LTF and R-LTF, using the workflow graph G of Fig. 2(a). Task execution times are $\mathcal{E}(t_1) = \mathcal{E}(t_7) = 15$, $\mathcal{E}(t_3) = 20$, $\mathcal{E}(t_2) = \mathcal{E}(t_6) = 6$ and $\mathcal{E}(t_4) = \mathcal{E}(t_5) = 5$. For simplicity, we assume that all edges have a cost of 2 time units to transfer a data item. We also assume a fully homogeneous network with $m = 8$ processors of speed $s = 1$. We let $\varepsilon = 1$ and $\mathcal{T} = 0.05$, so that the maximum allowed period is 20.

(i) **LTF scheduling steps:** At step 1, t_1 is the only ready task in $\alpha = \{t_1^{54}\}$, thus the chunk list $\beta = \{t_1^{54}\}$ (the superscript of a task in α or β denotes its priority value). t_1 is selected and scheduled on processors P_1 and P_5 (the

Algorithm 4.1 The LTF Algorithm

```
1: P = {P_1, P_2, ... P_m}; (*Set of processors*)
2: Δ ← 1/T iteration period;
3: Σ_u ← C_u^I ← C_u^O ← 0, ∀ 1 ≤ u ≤ m ;
4: ε ← maximum number of supported failures;
5: Compute bℓ(t) for each task t in G and set tℓ(t) = 0 for each entry task t;
6: S = ∅ ;  U = V ;  (*Mark all tasks as unscheduled*)
7: α = ∅ ;  (*List of ready tasks*)
8: Put entry tasks in α;
9: S ← 0; B ← m;
10: while U ≠ ∅ do
11:    k = 0; β ← ∅;
12:    while k ≤ B and α ≠ ∅ do
13:       β ← β ∪ H(α) ;  (*Select critical tasks *)
14:       P^k = ∅ ;  (*List of locked processors of t_k*)
15:       k = k + 1;
16:    end while
17:    for k = 0; k ≤ |β|; k++ do
18:       ∀ 1 ≤ i ≤ |Γ^-(t)|, compute λ_i;
19:       θ^k ← min_i(λ_i);
20:       Z^k = 0;
21:    end for
22:    for N = 0; N ≤ ε; N++ do
23:       for each task t_k ∈ β do
24:          if Z^k < θ^k then
25:             One-To-One-Mapping(t_k, P^k);
26:             Z^k = Z^k + 1;
27:          else
28:             F ← ∞;
29:             for each processor P_u ∈ P do
30:                if condition (1) is verified then
31:                   Compute F^u(t_k);
32:                   if (F^u(t_k) ≤ F) then
33:                      F ← F^u(t_k);
34:                      P(t_k) ← u;
35:                   end if
36:                end if
37:             end for
38:             Update S and P^k;
39:             Σ_P(t_k) ← Σ_P(t_k) + E(t_k)/s_u;
40:             Update C^I_P(t_k);
41:             Update C^O_P(t_*), t_* ∈ Γ^-(t_k), P(t_*) ≠ P(t_k);
42:          end if
43:       end for
44:    end for
45:    for each task t ∈ β do
46:       Put t in S and update priority values of its successors;
47:       Put ready successors of t in α;
48:       U ← U \ t;
49:    end for
50: end while
```

Algorithm 4.2 One-To-One-Mapping($t, \mathbb{P}$)

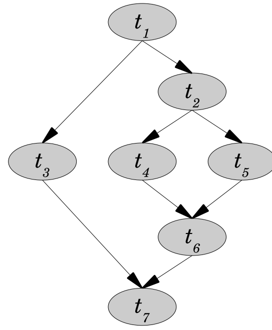
1: **for** $u = 0; u \le m; u++$ **do**
2: **if** condition 1 is verified **then**
3: $\forall\ 1 \le i \le |\Gamma^-(t)|$, sort the set $\overline{\mathcal{B}}(t_i)$ by non-decreasing order of their communication finish times $\mathcal{F}(c, l)$ on the links;
4: $T \leftarrow \bigcup_{1 \le i \le |\Gamma^-(t)|} \mathcal{H}\Big(\overline{\mathcal{B}}(t_i)\Big)$;
5: Simulate the mapping of t on processor P_u as well as the communications induced by the replicas of the set T to the links;
6: **end if**
7: **end for**
8: Select the (task, processor) pair that allows for the earliest finish time of t;
9: Schedule t onto the corresponding processor (call it P^*) and the incoming communications to the corresponding links;
10: Update $\mathcal{S}$;
11: $\Sigma_{P^*} \leftarrow \Sigma_{P^*} + \frac{\mathcal{E}(t)}{s_u}$;
12: Update $\mathcal{C}^I_{P(t_k)}$;
13: Update $\mathcal{C}^O_{P(t_*)}, t_* \in \Gamma^-(t_k), P(t_*) \neq P(t_k)$;
14: Update the set $\mathbb{P}$
$$\mathbb{P} \leftarrow \mathbb{P} \bigcup P^* \bigcup \left\{ \bigcup_{i=1}^{|\Gamma^-(t)|} P\Big(\mathcal{H}\Big(\overline{\mathcal{B}}(t_i)\Big)\Big) \right\}$$
15: Update each sorted list $\overline{\mathcal{B}}(t)$;
$\forall\ 1 \le i \le |\Gamma^-(t)|,\ \overline{\mathcal{B}}(t_i) \leftarrow \overline{\mathcal{B}}(t_i) \setminus \mathcal{H}\Big(\overline{\mathcal{B}}(t_i)\Big)$

task is replicated once to resist to one failure). At step 2, $\alpha = \{t_3^{54}, t_2^{53}\}$, $\beta = \{t_3^{54}, t_2^{53}\}$. t_2 and t_3 are scheduled in the order of their replicas $t_3^{(1)}$, $t_2^{(1)}$, $t_3^{(2)}$, $t_2^{(2)}$ on P_2, P_6, P_3 and P_7 according to condition (1) and their minimum finish times. At step 2, $\alpha = \{t_4^{53}, t_5^{53}\}$, $\beta = \{t_4^{53}, t_5^{53}\}$. Similarly the replicas of the two tasks are scheduled in the order on P_3, P_4, P_7 and P_8. At step 6, $\alpha = \{t_6^{53}\}$, $\beta = \{t_6^{53}\}$. The two replicas $t_6^{(1)}$ and $t_6^{(2)}$ of the task are scheduled on P_4 and P_8 respectively since these processors do not exceed the iteration period and allow for the minimum finish time of the task. After that step, we have $\Sigma_1 = \Sigma_2 = 15$, $\Sigma_2 = \Sigma_6 = 20$, $\Sigma_3 = \Sigma_7 = 10$, $\Sigma_4 = \Sigma_8 = 10$. So the remaining task t_7 cannot be scheduled without violating the desired throughput, and LTF fails to schedule the workflow. In fact, it needs two additional processors to succeed, as shown in figure 2(b): four pipeline stages are generated with 10 processors, and the latency is $\mathcal{L} = 140$.

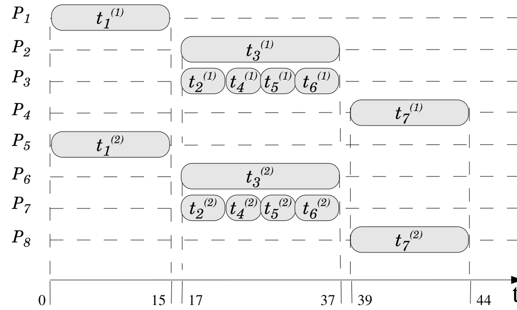
(ii) **R-LTF scheduling steps:** At step 1, t_7 is selected and scheduled on P_1 and P_5. Then $\alpha = \{t_3^{54}, t_6^{53}\}$, $\beta = \{t_3^{54}, t_6^{53}\}$ and $\mathcal{S} = 1$. At step 2, Rule 1 is not satisfied since none of the tasks can be merged with t_7. Therefore, according to Rule 2, all replicas are mapped on different processors P_2, P_3, P_6 and P_7 so that each replica will be assigned to a separate *singleton* processor (one-to-one mapping procedure). At steps 3 and 4, according to Rule 1, both $\{t_4, t_5\}$ and $\{t_2\}$ are mapped with t_6: the pipeline stage number $\mathcal{S} = 2$ does not increase. Finally, t_1 is selected and scheduled on P_4 and P_8. Three pipeline stages are generated. This results in a latency $\mathcal{L} = 100$ with 8 processors.

5 Experimental Results

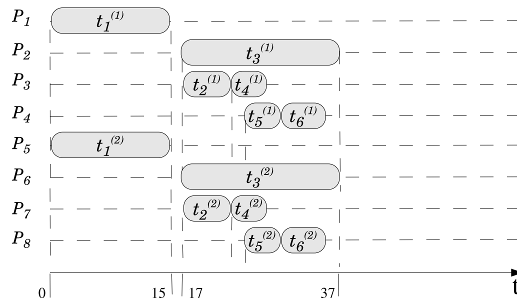
To evaluate the performance of our algorithms, several series of simulations have been conducted. We use randomly generated graphs, whose parameters are consistent with those used in the literature [1, 4, 11, 8]. The number of tasks is chosen uniformly from the range [50, 150]. The granularity of the task graph is varied from 0.2 to 2.0, with increments of 0.2. The number of processors is set to 20, the desired throughput is set to $\frac{1}{10(\varepsilon+1)}$ and we let $\varepsilon = \{1, 3\}$. To account for communication heterogeneity in the system, the unit message delay of the links and the message volume between



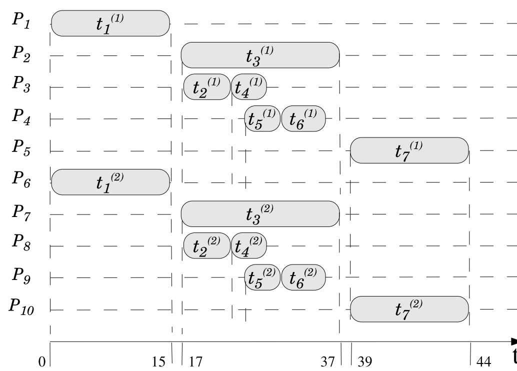
(a) - Workflow graph G



(b) - R-LTF schedule with m = 8



(c) - LTF schedule with m = 8 (LTF fails to schedule G)



(d) - LTF schedule with m = 10

Figure 2: LTF & R-LTF schedules

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. Each algorithm is evaluated in terms of achieved latency and fault tolerance overhead. We run algorithms LTF^c and R-LTF^c where the superscript c means that the resulting latency is the one achieved during an execution where c failures occur. When $c = 0$, we obtain LTF^0 and R-LTF^0: this corresponds to an execution where no failure has occurred, but with an algorithm designed to tolerate up to ε failures. We compare our algorithms to a reference schedule, the *fault free* schedule, defined as the schedule generated by R-LTF without replication, assuming that the system is completely safe, setting $\varepsilon = 0$. The overhead of each algorithm is computed as $\text{Overhead}_{\text{algo}} = \frac{\mathcal{L}_{\text{algo}} - \mathcal{L}_{\text{FF}}}{\mathcal{L}_{\text{FF}}}$, where $\mathcal{L}_{\text{algo}}$ is the latency achieved by the algorithm, and $\mathcal{L}_{\text{FF}}$ the latency of the fault free schedule.

Comparing the results of LTF and R-LTF, we observe in Figs. 3 and 4 that R-LTF gives the best performance. It always improves the latency significantly while meeting the throughput constraint. As stated above, R-LTF incrementally tries to decrease the pipeline stage number and communication overhead. This leads to minimize the final pipeline latency. The reason of the poorer performance of LTF can be explained by its processor selection policy: processors are selected so that the finish time of the tasks is minimized. Doing so, tasks are not mapped on those processors which would allow not to increase the pipeline stage number. 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 3(b) and 4(b) that R-LTF^c behaves better than LTF^c. As expected, LTF has a bigger latency.

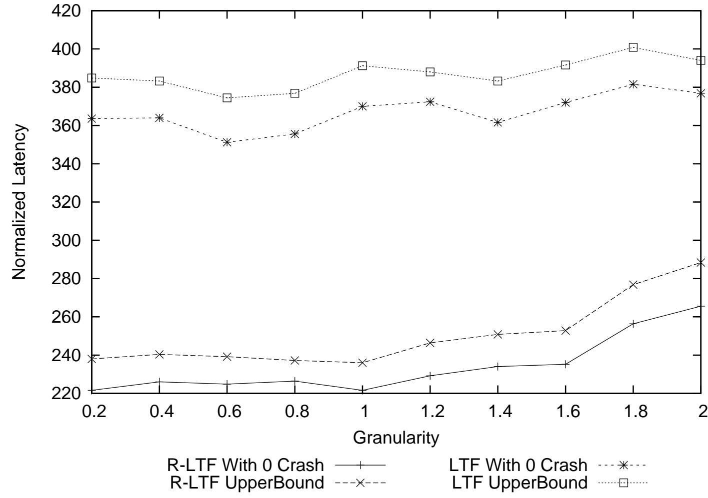
From Figures 3(b), it is interesting to note that when the fault tolerance degree is low ($\varepsilon = 1$), the latency is similar to that obtained with 0 crash (the lower bound). This is explained by the fact that the increase in the schedule length is already absorbed by the replication done previously, in order to resist to eventual failures. However, when the number of failures gets larger (for instance with $\varepsilon = 3$ and $c = 2$ failures, see Figure 4(b)), we clearly see the difference in terms of latency increase and overhead. We readily observe from Figures 3 and 4 that we deal with two conflicting objectives. Indeed, the fault tolerance overhead increases together with the number of supported failures.

As a summary of the experiments, we observe that R-LTF is considerably superior to LTF in all the cases tested $(0.2 \leq g(G) \leq 2, \varepsilon = \{1, 3\})$. We also state that the pipeline stage number has a significant impact on the latency achieved by LTF. This experimental study assesses the usefulness of R-LTF, and shows that reducing the pipeline stage number should be given priority to minimizing communication overhead.

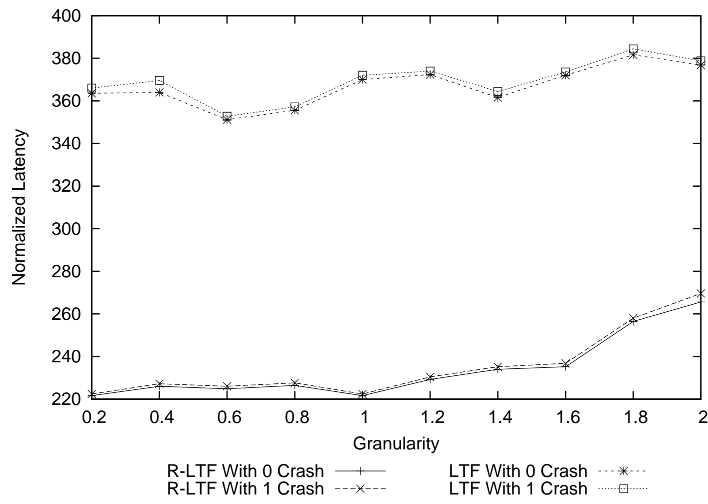
6 Conclusion

In this paper, we have addressed the problem of multi-criteria scheduling for workflow applications. This a very natural and important problem, as several conflicting objectives must be considered simultaneously to fulfill the requirements of the user. We have selected three out of the most prominent criteria, two performance-related (throughput and latency), and one reliability-oriented (resisting to several processor failures). To the best of our knowledge, the proposed algorithms are the first to address such a challenging tri-criteria optimization problem, using realistic platform models.

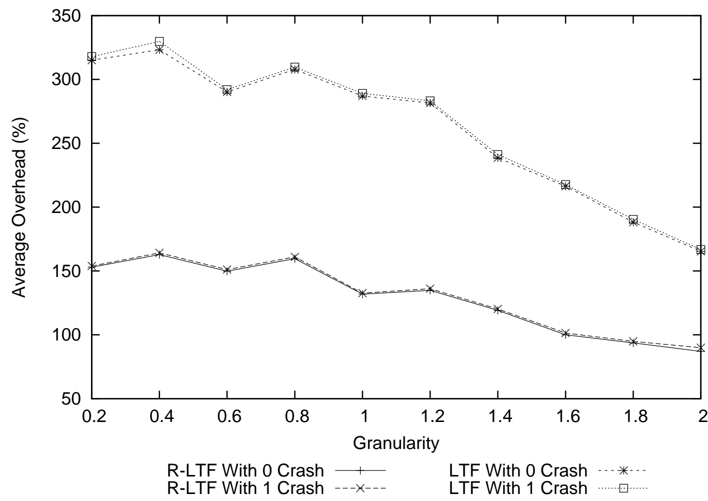
Our approach should be extended to situations "symmetric" to that of this paper, namely maximizing the throughput for a given latency and failure number, and maximizing the number of supported failures for a given latency and throughput. Further work will also be devoted to designing algorithms involving other important objectives, such as energy consumption (e.g., minimize the dissipated power for a prescribed performance) and platform cost (e.g., minimize the 'rental" cost of the platform while enforcing the other criteria).



(a) - Latency bounds ($\varepsilon = 1$)

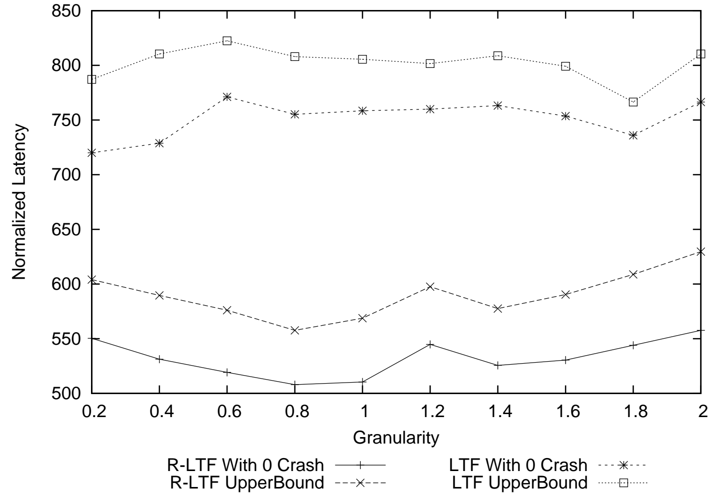


(b) - Latency with crash ($\varepsilon = 1$)

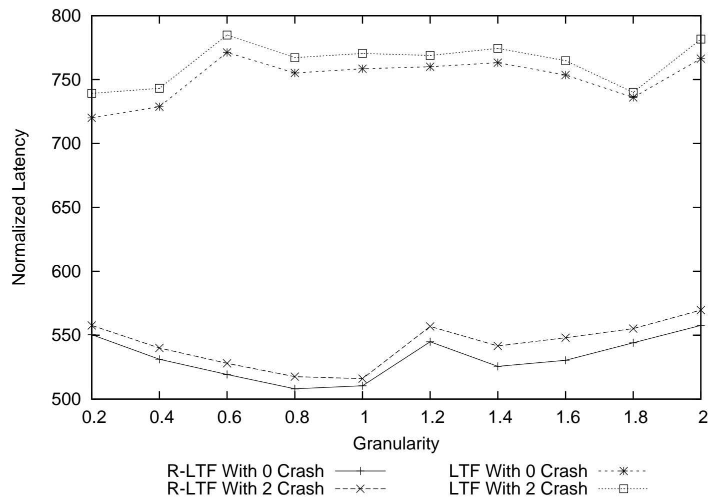


(c) - Fault tolerance overhead with crash ($\varepsilon = 1$)

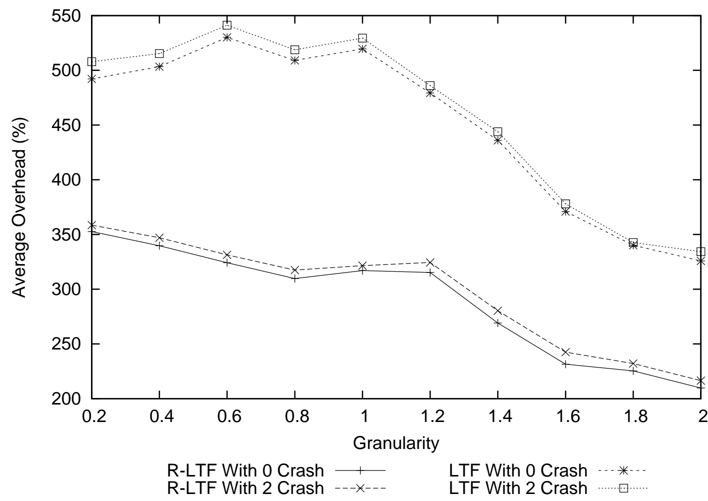
Figure 3: Average normalized latency comparison between LTF and R-LTF (Bound and Crash cases, $\varepsilon = 1$)



(a) - Latency bounds ($\varepsilon = 3$)



(b) - Latency with crash ($\varepsilon = 3$)



(c) - Fault tolerance overhead with crash ($\varepsilon = 3$)

Figure 4: Average normalized latency comparison between LTF and R-LTF (Bound and Crash cases, $\varepsilon = 3$)

References

[1] A. Benoit, M. Hakem, and Y. Robert. Contention awareness and fault tolerant scheduling for precedence constrained tasks on heterogeneous systems. *Parallel Computing*, 35(2):83–108, 2009.

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

[3] F. Guirado, A. Ripoll, C. Roig, and E. Luque. Optimizing latency under throughput requirements for streaming applications on cluster execution. In *Cluster Computing*, pages 1–10. IEEE Computer Society Press, 2005.

[4] S. L. Hary and F. Ozguner. Precedence-constrained task allocation onto point-to-point networks for pipelined execution. *IEEE Trans; Parallel and Distributed Systems*, 10(8):838–851, 1999.

[5] P. D. Hoang and J. M. Rabaey. Scheduling of dsp programs onto multiprocessors for maximum throughput. *IEEE Trans. Signal Processing*, 41(6):2225–2235, 1993.

[6] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, 1989.

[7] K. Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Trans. Parallel and Distributed Systems*, 6(4):412–420, 1995.

[8] S. Ranaweera and D. P. Agrawal. Scheduling of periodic time critical applications for pipelined execution on heterogeneous systems. In *Int. Conf. Parallel Processing ICPP'01*, pages 131–140. IEEE Computer Society Press, 2001.

[9] 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.

[10] N. Vydyanathan, U. Catalyurek, T. Kurc, P. Saddayappan, and J. Saltz. Toward optimizing latency under throughput constraints for application workflows on clusters. In *Euro-Par'07: Parallel Processing*, LNCS 4641, pages 173–183. Springer Verlag, 2007.

[11] M.-T. Yang, R. Kasturi, and A. Sivasubramaniam. A pipeline-based approach for scheduling video processing algorithms on now. *IEEE Trans. Parallel and Distributed Systems*, 4(2):119–130, 2003.