HAL
open science

Dynamic Adaptation Applied to Sabotage Tolerance

Serge Guelton, Thierry Gautier, Jean-Louis Pazat, Sébastien Varette

To cite this version:

Serge Guelton, Thierry Gautier, Jean-Louis Pazat, Sébastien Varette. Dynamic Adaptation Applied to Sabotage Tolerance. [Research Report] RR-6659, INRIA. 2008. inria-00323226

HAL Id: inria-00323226

https://inria.hal.science/inria-00323226v1

Submitted on 4 Oct 2008

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

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

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Dynamic Adaptation Applied to Sabotage Tolerance

Serge Guelton — Thierry Gautier — Jean-Louis Pazat — Sébastien Varette

N° 12345

2008 September

Thème NUM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--12345--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche GRENOBLE - RHÔNE-ALPES

Dynamic Adaptation Applied to Sabotage Tolerance

Serge Guelton* , Thierry Gautier† , Jean-Louis Pazat* , Sébastien Varette‡

Thème NUM — Systèmes numériques
Équipes-Projets Paris and Moais

Rapport de recherche n° 12345 — 2008 September — 14 pages

Abstract: Distributed computing platforms contribute for a large part to some of the most powerful computers. Such architectures raise new challenges, typically in terms of scheduling, adaptability and security. This paper addresses the issue of result-checking in distributed environments, where tasks or their results could have been corrupted due to benign or malicious acts. Using a macro-data flow representation of the program execution, this article presents a novel approach based on work-stealing scheduling to dynamically adapt the execution to sabotage while keeping a reasonable slowdown rate. Unlike static adaptation or adaptation at the source code level, a dynamic adaptation at the middleware level is proposed, enforcing separation of concepts and programming transparency. This article contains both conceptual and experimental results that show the interest, feasibility and limits of the concept.

Key-words: Dynamic adaptation, work-stealing, sabotage tolerance

* Paris research team, Rennes University, France Email: Firstname.Lastname@irisa.fr
† Moais research team, LIG Laboratory, France Email: Thierry.Gautier@imag.fr
‡ CSC research unit, University of Luxembourg, Luxembourg Email: Sebastien.Varrette@uni.lu

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Adaptation Dynamique appliquée à la tolérance aux fautes

Résumé : Les plateformes de calcul distribué contribuent dans une large proportion à la plupart des machines de calcul les plus puissantes. De telles architectures font apparaître de nouveaux défis, typiquement en termes d'ordonnancement, d'adaptabilité et de sécurité. Ce document répond au problème de vérification des résultats en environnent distribué, où les tâches ou leurs résultats peuvent être corrompus suite à des attaques ou des erreurs. Grâce à une représentation de l'exécution du programme sous forme de graphe de flot de données, l'article présente une approche novatrice basée sur un ordonnancement par vol de tâche pour adapter dynamiquement l'exécution à la falsification de résultats tout en conservant une perte de performance raisonnable. Contrairement aux méthodes d'adaptation statique ou au niveau du code source, une adaptation dynamique au niveau de l'intergiciel est proposé, offrant ainsi une bonne séparation des concepts et une plus grande clarté de programmation. L'article contient à la fois des aspects théorique et des résultats expérimentaux qui montrent l'intérêt, la faisabilité et les limites du concept proposé.

Mots-clés : Adaptation Dynamique, vol de tâches, tolérance aux fautes

1 Introduction

Large scale computing systems such as grids and peer-to-peer computing platforms gather thousands of resources for executing parallel applications. Even if a middleware is used to secure the communications and to manage the resources, the computational nodes operate in an unbounded environment and are subject to a wide range of attacks able to alter the computed results [**?**, **?**]. Such attacks are especially of concern due to Distributed Denial of Service (DDoS), virus or trojan attacks, and more generally orchestrated attacks against widespread vulnerabilities of a specific operating system. Corruption of computed results, called sabotage or cheats in the literature, remains a fundamental issue for parallel executions as a potentially large part of the computation can be lost just because it operates on inputs that have been altered. It should be reminded that this is not purely fiction: whether the attacks conducted to alter results are the consequences of a malicious act or not, they still have been experimented in SETI@Home [**?**, **?**] or more recently in BOINC [**?**].

The presence of saboteurs in grid computing systems is well-known and leads to result-checking algorithms. Many countermeasures have been proposed in literature [**?**, **?**, **?**, **?**, **?**]. Yet, to find wrong results created by lazy participants or malicious cheaters, the only generic approach relies on tasks duplication, either total or partial. This scheme generally assumes the execution domain to be partitioned in two area R (resp. U), each having trusted/reliable (resp. untrusted/unreliable) resources (see fig 1). The first kind of resources is used to conduct safe re-executions (*i.e.* duplications) on verifiers in order to certify *i.e.* build a trust level on the results computed over the second kind of resources.

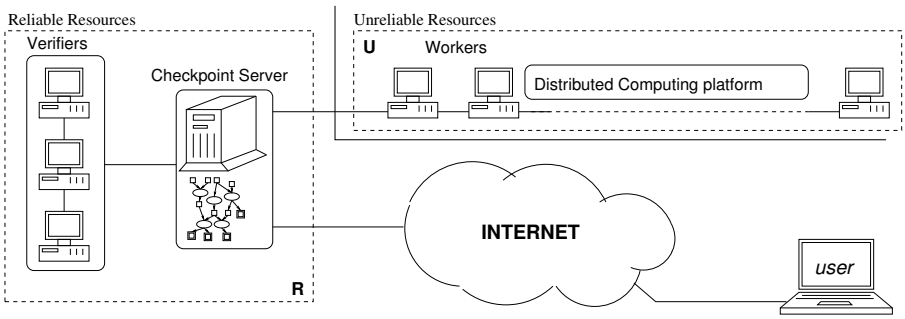


Figure 1: Configuration of a computing platform able to handle both the execution of a parallel program and its certification [**?**].

In [**?**], a set of certification algorithms based on partial task duplication and macro data flow analysis have been proposed to tackle the issue of massive attacks in which the number of corrupted results exceed a given threshold.

Currently, the proposed approach suffers two problems:

- it requires a deep integration with user code, thus not offering good programming transparency;
- the user must find a good balance between the number of certification nodes, the slowdown rate and the quality of the certification.

The work presented in this paper extends such approaches and describe an integrated sabotage-tolerance mechanism which does not suffer the first issue, and delegates the resolution of the second to a decision tool. More precisely, our proposal makes use of the work-stealing scheduling based on macro-data flow analysis implemented in the KAAPI middleware [**?**] to adapt the task flow through the DYNACO framework [**?**].

The article is organized as follows. Section 2 briefly recall related works while sections 3 and 4 describe the two underlying tools used in our approach, namely KAAPI and DYNACO . In particular, we will see in section 5 why KAAPI suits well to dynamic adaptation through the use of particular adaptation point [**?**]. This will lead to the definition of a novel certification approach called *Thief Induced Certification*. Afterwards an implementation of an adaptation framework for task based parallelism is described in section 6, followed by some experimentations in section 7. Section 8 will finally conclude and propose some opening for future works.

2 Related works

This paper is at the crossroad of three fields: dynamic adaptation, sabotage-tolerance and work-stealing based distributed computing.

2.1 Dynamic adaptation

Adaptation of existing software is a well known concern. The goal is to give a program the ability to react to change of its environment. Dynamically adapting distributed programs goes a step further, for you may need to guarantee a kind of synchronization between adaptations.

- Buisson & al. in [**?**] focuses on the analysis of the adaptation scheme. They validated their approach on both sequential and parallel programs, but gave no clue concerning how to manage coordination between adapted nodes.
- Camara & al. in [**?**] tackles the interest of Aspect Oriented Programming (AOP) to manage dynamic adaptation.
- Truyen & al. in [**?**] view dynamic adaptation as a set of dynamic aspect weaving. Enforcing the need for coordination support, they provide a protocol for coordinated reconfiguration.

Nevertheless, these solutions all rely on dynamic aspect weaving to insert interceptors inside user code. Aspect weaving support has reach industrial level for a long time for the Java language, but is still incomplete for C++. AspectC++ [**?**] does not support template code, and its dynamic weaving functionality is not implemented yet. DAC++ [**?**] is another aspect weaver for C++ that supports dynamic weaving, as well as network aspects. Yet its weaving capabilities are also limited to a subset of object oriented features of the C++ language. As a consequence, AOP does not seem to be the right way to go to.

2.2 Sabotage tolerance

As mentioned in the introduction, distributed computations are subjected to various attacks, some of them corrupting intermediate results. Two complementary strategies could then be used:

1. preventing *a priori* the forgeries by making them more difficult to perpetrate. This can be achieved by reinforcing the global security of the architecture using for instance quotas on CPU/memory, firewalls, sandboxing techniques, monitoring, etc. It is also possible to check formally the correctness of the program before the execution [**?**], typically through Proof-Carrying Codes [**?**] or an assembly language like TAL [**?**]. Yet, this approach is still expensive and does not increase the reliability of the computed results in distributed environment where the computing resources cannot be fully trusted;

2. controlling *a posteriori* the output of the tasks by evaluating their correctness. This article focus on this strategy and a brief state-of-the-art relative to this approach is now provided.

At first, one can check the program behavior for some well-known input/output pairs [**?**] as in classical challenge/response procedures. One issue is then to maintain the diversity on the challenge tasks such that this mechanism cannot be fouled by a clever attacker able to recognize them.

Another field of research is dedicated to property checking [**?**] derived from the notion of *simple checkers* introduced in [**?**]. The idea is that for some problems, the time required to carry out the computation is asymptotically greater than the time required to determine whether or not a given result is correct. This is possible thanks to a post-condition the output have to conform. If this approach remains very simple and elegant, it is often impossible to automatically extract such post-condition on a program.

Generic approaches are based on duplication. At this level, previous works [**?**, **?**] consider successive *batchs* composed of n independent tasks. Under those hypothesis, two mechanisms are proposed in the literature:

- Germain & al. in [**?**] try to certify as soon as possible the quality of the batch in the framework of statistical testing. More precisely, a probabilistic test based on sequential analysis is proposed to certifies the batch in an adaptive way, without unduly eliminating results which are actually correct and with a relatively low cost (defined as the number of calls to oracles, the entity used to re-execute tasks).

- Sarmenta in [**?**] enforces the batch quality by making the falsification probability lower than the tolerance threshold δ of the application. Concepts like voting, spot-checking, blacklisting or credibility-based fault-tolerance are then applied to decrease iteratively the error rate of the batch.

Both approaches are limited to the restricted context of independent tasks with a modeling of attacker behavior. Probabilistic techniques for direct certification have been extended to any parallel computation with potentially dependent tasks in [**?**, **?**, **?**, **?**]. In addition, no particular assumption on the attack nor on the distribution of errors have been made by the authors which make the

certification algorithms particularly attractive. Proposed algorithms relies on a portable representation for the distributed execution of a parallel program over fixed inputs: a bipartite Direct Acyclic Graph $G = (\mathcal{V}, \mathcal{E})$ known as a *macro-data flow graph*. More precisely, the MCT algorithm (*Monte Carlo Test*) that makes use of the graph to certify efficiently applications composed of independent tasks. Dependencies issues are tackle by the $EMCT$ algorithm (*Extended Monte Carlo Test*) and its variant. A precise cost analysis of the certification algorithms assuming an on-line scheduling by work-stealing has been done in [**?**], proving the low overhead induced by the result-checking for specific classes of graphs (and therefore of programs), namely trees and Fork-Join graphs. We will have the opportunity to reuse those results in this paper.

2.3 Dynamic scheduling by work-stealing

To support efficiently the execution of fully-strict series-parallel programs [**?**], we assume the execution engine to implement an on-line scheduling by work stealing following the work-first principle. The principle is simple. Each processor serially executes the tasks it has locally created according to a depth-first order. When a processor becomes idle, it steals the oldest ready task (breadth first order) on a non-idle processor which is randomly chosen in general. This approach is implemented by the parallel programming interfaces Cilk [**?**, **?**] and KAAPI [**?**]. In particular, KAAPI supports processors with changing speeds and volatility [**?**]. As we use this last middleware for our implementation, more details about KAAPI will be given in section 4.

Following [**?**, **?**], we assume a bounded ratio between the fastest and the slowest participating processors of the computing grid. Let Π and Π^{tot} be respectively the average speed (number of unit operations per second) per processor and the total average speed (e.g., assuming n_p processors, $\Pi^{tot} = n_p\Pi$). Let W_1 be the total work (number of unit operations) of the parallel program to execute and W_∞ its depth (maximal number of unit operations on a critical path) on an unbounded number of processors. Then, from Theorem 6 in [**?**], the program is scheduled with high probability in time

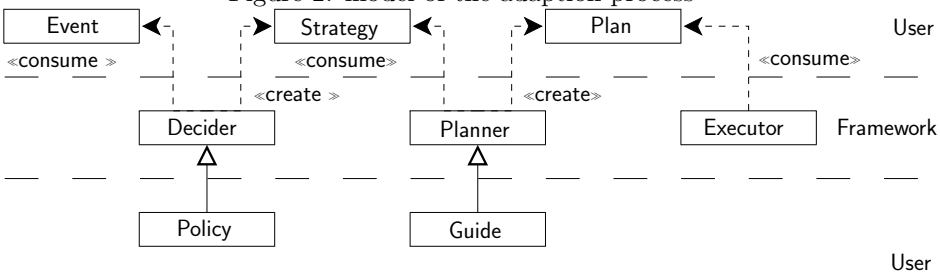
$$T \leq \frac{W_1}{\Pi^{tot}} + \mathcal{O}\left(\frac{W_\infty}{\Pi}\right)$$

This paper combine the three domains presented in this section to propose a concrete framework that integrate transparently certification algorithms. To our knowledge the combination of these three fields has not been studied yet.

3 Dynaco

Adaptation is a fuzzy concept : it is more or less admitted that an adaptation is the possibility to react to an event, self generated or not. For example, adding or removing nodes during a parallel run in order to increase the computation efficiency or to stop suspicious nodes is an adaptation. In the following, we will consider such adaptations. The change needed to give adaptable aspect to a program are not easy to formalize. J. Buisson proposes in [**?**] a model for adaptation of distributed components. This model is bundled with a Java

Figure 2: model of the adaption process



framework based on the Fractal component model and its Julia implementation [**?**]. It helps to add adaptation capabilities to a program while keeping the adaptation code segregated from the user code. He introduces the concept of adaptation points, particular places in the program where an adaptation can be performed. The binding between user code and adaptation code is held in these particular points, often through the use of aspect weaving techniques. Figure 2 summarize the steps involved during adaptation:

1. an Event is generated by a probe, for example "node A failed to answer our challenge";
2. a Decider receives event and make decision;
3. a Strategy is issued by the Decider , for example "banish node A";
4. a Planner planify the action for a Strategy;
5. a Plan is issued by the Planner, for example "stop the whole computation, then blacklist the IP of A and restart";
6. an Executor executes the actions from the plan back to the adapted component.

The Decider and Planner are dependent from the adaptation and provided by the user as Policy and Guides, while the adaptation engine is provided by the DYNACO framework. Link between the framework and the user code is done during the component assembling.

4 Kaapi

KAAPI stands for Kernel for Adaptive, Asynchronous Parallel and Interactive programming. It is a middleware implemented as a C++ library that let user run programs described as Direct Acyclic Graph (DAG)s – a set of tasks and data dependencies. KAAPI uses this description to schedule the user tasks with respect to data dependencies using a work-stealing algorithm [**?**]. More accurately, each computing node owns a set of thread, each of which has a local dequeue of tasks. Data dependencies are given as different access rights that a task can take to variables put in the shared memory. Load balancing is done *via* the steal algorithm: when the dequeue of a thread is empty, it performs a local steal

Figure 3: remote steal sequence

```
#include <athapascan-1>

void sum(shared r int res1,
         shared r int res2,
         shared w int res)
{ res = res1+res2; }

void fibonacci( int n, shared w int res )
{
 if (n < 2) res = n;
 else {
   shared int res1;
   shared int res2;
   fork fibonacci(n-1, res1 );
   fork fibonacci(n-2, res2 );
   fork sum( res1, res2, res );
 }
}
```

request on local threads, followed by a remote steal request on other node in case of failure. For recursive algorithms with good data locality, this scheduling proves to be efficient[**?**]. One advantage of this algorithm is that nodes only communicate data through the steal operation. When a task is stolen, the input data are transfered , and when the stolen task ends, the output data are written back. This property will be further used in section 5.

Figure 3 shows a sample code written in the Athapascan language. `fork` s are used to spawn new tasks and `shared` s are used to put data in the shared memory where they can be accessed with various right.

5 Thief Induced Certification

In this section, we will show how DYNACO and KAAPI can be used to add certification *via* replication to a parallel program. In section 4 we showed that the steal method is the only kind of communication between nodes. In section 3 we pointed out the need for adaptation points. Indeed the steal methods is the right place to insert replication: as long as the execution is performed on a trusted node, there is no need for replication, but as soon as a steal request comes from a potentially malicious node, replication should occur.

Using the low level Application Programming Interface (API) of KAAPI , we directly transform the DAG to add both replication and certification. The steal scenario is modified as described in figure 4.

Using the introspection API of KAAPI , we only duplicate tasks that write into shared memory, other are needless to check. Note that we make sure that the duplicated task and the certification task are not stealable: they will only be executed on a trusted node.

remote steal sequence

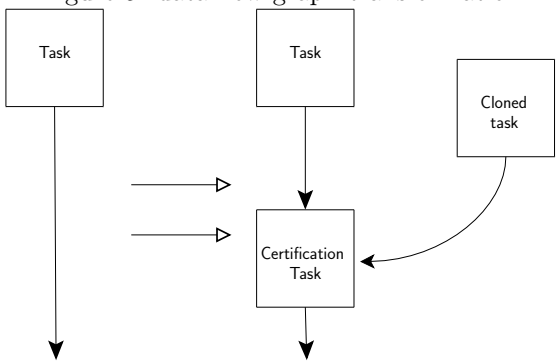
```
steal request
find ready task in victim
atomically set task to
    stolen
copy task to thief

...
```

remote steal with certification sequence

```
steal request
find ready task in victim
atomically set task to
    stolen
copy task to thief
duplicate task in victim
add certification task in
    victim
```

Figure 4: modifying remote steal sequence

Figure 5: data flow graph transformation



The transformation is summarized in figure 5. This transformation is done using the reification [**?**] of the steal method: the changes of the middleware are deported to the user level. This would still require some code snippet inside user code, so we added a module loader to load the reification module at runtime. Thanks to these mechanisms, the adaptation point can be inserted without modifying a single line of code neither in the user program nor in the middleware, achieving our first objective.

The remaining problem is that if we make the duplication each time a task is stolen, the execution time will be greater than the sequential time, each task being executed once on the trusted node. Consequently, as proposed in [**?**], we use and compare several certification techniques: this will be the core of our adaptation for the sabotage-tolerance aspect.

6 Sketch of the adaptation

Considering a parallel run, we again assume two execution area : the trusted one R where the computation can be reliably and safely done, and the untrusted area U where some computation can be corrupted. In the sequel, we still assume that nodes never falsify their identity. The program execution over KAAPI is located in both area, while the DYNACO - based certification engine is located in the

trusted area. All communication between the trusted and untrusted area are done through the steal method, and require the acknowledgment of the certifier.

Each time a steal begins, the certifier is called, and depending on its current state, it requires the certification of the stolen task. The steal proceeds and the certification is done in parallel (depending on available resources). If the certification succeeds a success event is sent and proceeded by the certifier. If the certification fails, the whole computation is stopped. The study of the more complex recovery mechanisms involving checkpointing is beyond the scope of this article.

We have implemented the routines to automatically add certification tasks into the data flow graph, the certifier and the steal reification. Therefore the result-checking process is kept transparent to the user. Several certification scenarii have been considered. They are now described.

6.1 k-check-never

In this model, The descision framework is asked for the need of certification, but it will always reply unfavorably. It is used as a witness experiment for further models.

6.2 l-greylisting

In this model, untrusted nodes computations are considered reliable only after a fixed number of success, namely l.

6.3 Monte-Carlo Test

Finally, it is possible to apply the certification algorithms Monte-Carlo Test (MCT) mentioned in section 2.2. The main idea is to randomly check tasks among those of the DAG. As given in [**?**], the number of checks only depends on the error probability the user is ready to accept together with the minimal number of sabotage assumed (in fact, the ration between this number and the total number of tasks)

In all cases, using the ability to easily modify the deployment descriptor of DYNACO , we experimented these simple certification strategies. Results are presented in next section.

7 Experiments

The following graph shows the performance impact of the certification using various strategies. The experimentation domain is grid5000 with up to 100 nodes connected to perform the computation. The test program, shown in figure 6 is a simple, one level fork-join loop with various certification strategies. All communication between the user program (in c++) and the certifier (in Java) are done through a CORBA bus.

As we can see in figure 7, such an application do not suffer from the centralization of steal request. It is mainly due to the fact that only calls to certification nodes are caught. We also see that using a single certification nodes make the critical path on this node much longer, thus decreasing performances.

```
#include <athapascan-1>

void compute(int i, shared w int res)
{ res = some_func(i); }

void print(int i, shared r int res)
{ std::cout << "i: " << res << std::endl; }

/* ... */
for( int i=0;i< n; i++)
{
    shared int res;
    fork compute(i,res);
    fork print(i, res);
}
```

Figure 6: simple fork join program

Figure 7: comparison of various certification strategies

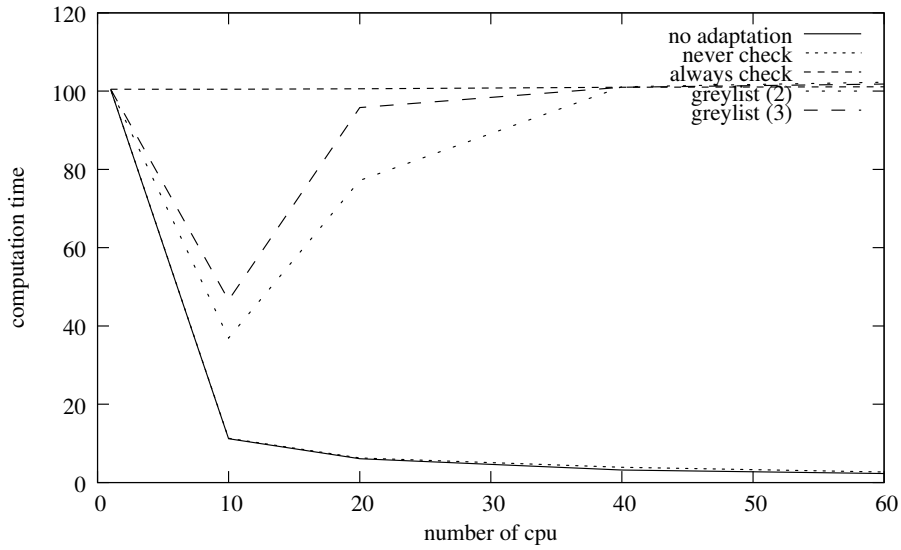
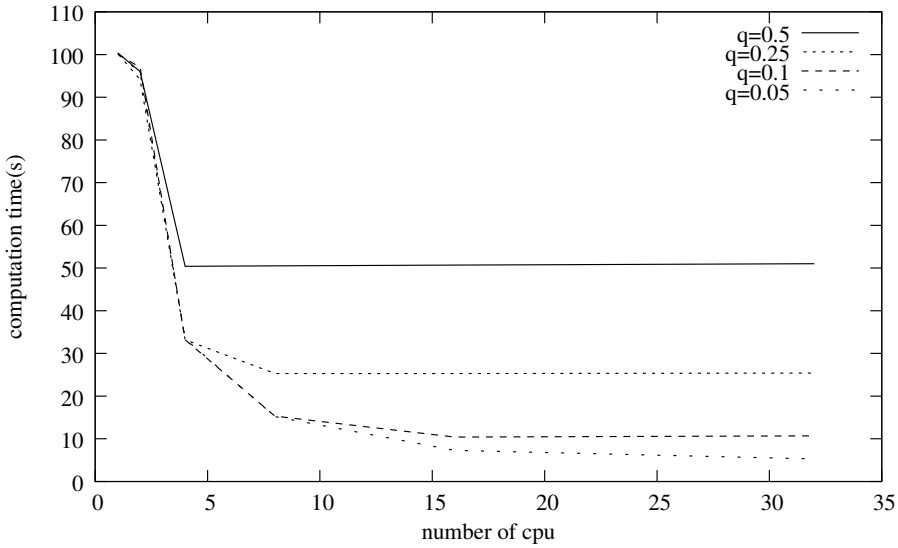


Figure 8: influence of the certification rate in MCT



A more interesting decider uses a MCT guide, with pre-defined certification rate. In our configuration, if we neglect communication time and if T_{task} denotes the execution time of a single task, N_{worker} the number of worker (including the certifier), N_{task} the number of independent tasks and q the certification rate, we get equation 1 for the certification time T_{certif} and equation 2 for execution time T_{exec}.

$$T_{certif} = T_{task} * N_{task} * q \tag{1}$$

$$T_{exec} = T_{certif} + max(0, N_{task} * \frac{1 - q * (N_{worker} - 1)}{N_{worker}}) \tag{2}$$

This guide has been inserted in the Dynaco framework to achieve results given in figure 8 that confirms the equation 2.

The MCT guide itself could be adapted in such a way that the certification rate would vary during execution.

8 Conclusions

In this article we presented a methodology to add adaptable sabotage-tolerance aspect to a distributed program. Through the use of adapted tools, we achieved the separation of concerns, segregating the user code, the adaptation and the adaptation point in three different entities. As a result, though a large panel of technologies is used, a reusable set of tools and libraries make further development easier.

Despite its adaptability, we still found limitations in the KAAPI middle ware, for it offers no support for secure authentication: a node is identified by its global id, a single number, and this is easily falsified.

Future development could be the modification of the certification technique to provide election: instead of making only one local clone of the stolen task, you prepare n clones that are not flagged as local. The certification tasks are then replaced by election tasks, taking n values as input and outputting the elected value. These tasks would still to be executed on a trusted node.

Contents

1 Introduction **3**

2 Related works **4**
2.1 Dynamic adaptation 4
2.2 Sabotage tolerance 5
2.3 Dynamic scheduling by work-stealing 6

3 Dynaco **6**

4 Kaapi **7**

5 Thief Induced Certification **8**

6 Sketch of the adaptation **9**
6.1 k-check-never 10
6.2 l-greylisting 10
6.3 Monte-Carlo Test 10

7 Experiments **10**

8 Conclusions **12**

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

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

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