Defining and Controlling the Heterogeneity of a Cluster: the Wrekavoc Tool

Louis-Claude Canon, Olivier Dubuisson, Jens Gustedt, Emmanuel Jeannot

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Defining and Controlling the Heterogeneity of a Cluster: the Wrekavoc Tool

Louis-Claude Canon — Olivier Dubuisson — Jens Gustedt — Emmanuel Jeannot

N° 7135

December 2009

Domaine 3

ISSN 0249-6399 ISRN INRIA/RR--7135--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
NANCY - GRAND EST

Defining and Controlling the Heterogeneity of a Cluster: the Wrekavoc Tool

Louis-Claude Canon[*† ‡] , Olivier Dubuisson[§†] , Jens Gustedt[†] ,
Emmanuel Jeannot[†‡]

Domaine : Réseaux, systèmes et services, calcul distribué
Équipes-Projets AlGorille and Runtime

Rapport de recherche n° 7135 — December 2009 — 38 pages

Abstract: The experimental validation and the testing of solutions that are designed for heterogeneous environments is challenging. We introduce Wrekavoc as an accurate tool for this purpose: it runs unmodified applications on emulated multisite heterogeneous platforms. Its principal technique consists in downgrading the performance of the platform characteristics in a prescribed way. The platform characteristics include the compute nodes themselves (CPU and memory) and the interconnection network for which a controlled overlay network above the homogeneous cluster is built. In this article we describe the tool, its performance, its accuracy and its scalability. Results show that Wrekavoc is a very versatile tool that is useful to perform high-quality experiments (in terms of reproducibility, realism, control, etc.)

Key-words: Tool for experimentation, performance modeling, emulation, heterogeneous systems

This paper has been accepted for publication Journal of Systems and Software.

[*] Nancy University
[†] AlGorille Team, INRIA Nancy – Grand Est
[‡] Runtime Team, INRIA Bordeaux – Sud-Ouest
[§] Felix Informatique

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
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Définir et contrôler l'hétérogénéïté d'une grappe : l'outil Wrekavoc

Résumé : La validation expérimentale et le test de solutions qui ont été conçues pour des environnements hétérogènes est un véritable défi. À cette fin, nous introduisons Wrekavoc, dont l'objectif est de répondre à ce problème de manière précise, en exécutant des applications non modifiées sur des plates-formes multi-sites hétérogènes émulées. La principale technique employée consiste à dégrader de manière prédéfinie les caractéristiques de la plate-forme utilisée. Les caractéristiques concernées sont : les nœuds de calcul eux-mêmes (CPU et mémoire) et le réseau d'interconnexion pour lequel un overlay est définit et construit au dessus de la grappe homogène considérée. Dans cet article nous décrivons l'outil, sa performance, sa précision et son extensibilité. Les résultats montrent que Wrekavoc est un outil très versatile qui est utile pour effectuer des expériences de haute qualité en termes de reproductibilité, réalisme, contrôle, etc.

Mots-clés : outil pour l'expérience, modélisation de la performance, émulation, systèmes hétérogènes

1 Introduction

Distributed computing and distributed systems is a branch of computer science that has recently gained very large attention. Grids [1], clusters of clusters [2], peer-to-peer systems [3, 4], desktop environments [5, 6], are examples of successful environments on which applications (scientific, data managements, etc.) are executed routinely.

However, such environments are composed of a multitude of different elements that make them more and more complex. The hardware (from CPU cores to interconnected clusters) is hierarchical and heterogeneous. Programs that are executed on these infrastructures can be composite and extremely elaborate. Huge amounts of data, possibly scattered on different sites, are processed. Numerous protocols are used to inter-operate the different parts of these environments. Networks that interconnect the different hardware are also heterogeneous and multi-protocol.

As a consequence, applications (and the algorithms implemented by them) are equally complex and become very hard to validate. However, validation is of key importance for application software: it assesses the correctness and the efficiency of the proposed solution, and allows for the comparison of a given solution to other already existing ones. Analytic validation consists in modeling the problem space, the environment and the solution. Its goal is then to gather knowledge about the modeled behavior using mathematics. For the domain that we are investigating here, analytic validation is often infeasible due to the complexity and partial unpredictability of the studied objects.

It is therefore mandatory to switch to experimental validation. This consists in executing the application (or a model of it), observing its behavior on different cases and comparing it with other solutions. This necessity for experiments truly makes this field of computer science an *experimental science*.

As in every experimental science, experiments are made through the means of tools and instruments. In computer science one can distinguish different methodologies for performing experiments, namely, benchmarking, real-scale, simulation and emulation [7].

Here, we describe a new emulator called Wrekavoc. The goal of Wrekavoc is to transform a homogeneous cluster into a multi-site distributed heterogeneous environment. This is achieved by degrading the perceived performance of the hardware by means of software that is run at user level. Then, using this emulated environment a real unmodified program can be executed to test and compare it with other solutions. Building such a tool is a scientific challenge: it requires to establish links between reality and models. Such models need to be validated in order to understand their limits and to assess their realism. However, a brief look at the literature shows that concerning simulators [8, 9] or emulators [10, 11], the validation of the proposed tools (end hence the models used in them) as a whole is seldom addressed.

The contributions of this paper are as follows. First, we present Wrekavoc, our tool. We describe its features, the model of configuration and some implementation details. Then, in an intensive experimental campaign, we demonstrate the *realism* of Wrekavoc. In order to do that, we first run a suite of micro-benchmarks to evaluate each proposed features independently. Second, we compare the execution of different applications on a heterogeneous platform with the execution on a homogeneous cluster and Wrekavoc.

This is done by using many different parallel programming paradigms and by executing exactly the same applications on the real platform and in the emulator. Last, we assess the scalability of Wrekavoc by using a large number of nodes (up to 200). Based on these experiments, we then conclude that our emulation tool called Wrekavoc is able to help in experimentally validating a solution designed for a distributed environment.

2 Related work

Here, we review some tools described in the literature that allow to perform large scale grid experiments. None of these tools allows the execution of an unrestricted and unmodified application under precise and reproducible experimental conditions that would correspond to a given heterogeneous environment. For a more complete survey of large-scale experiment environment the reader is referred to [7].

2.1 Real-scale Experimental Testbeds

Grid'5000 [2] is a national French initiative to acquire and interconnect clusters on 9 different sites into a large testbed. It allows for experiments to run at all levels from the network protocols to the applications. This testbed includes Grid Explorer [12], a designated scientific instrument with more than 500 processors.

Das-3 [13], the Distributed ASCI Supercomputer 3, is a Dutch testbed that links together 5 clusters at 5 different sites. Its goal is to provide infrastructure for research in distributed and grid computing.

Grid'5000 and Das 3 have very similar goals and collaborate closely. They are connected by a dedicated network link.

Planet-lab [14] is a globally distributed platform of about 500 nodes, all completely virtualized. It allows the deployment of services on a planetary scale. Unfortunately, its dynamic architecture makes the controlled reproduction of experiments difficult.

These platforms allow the benchmarking of any type of application. Nevertheless, each platform by itself is quite homogeneous and thus the control and the extrapolation of experimental observations to real distributed production environments is often quite limited. In addition, the management of experimental campaigns is still a tedious and time-consuming task.

2.2 Simulators

Bricks [15], SimGrid [8] and GridSim [9] are simulators that allow the experimentation of distributed algorithms and the study of the impact of platforms and their topology. In particular, these simulators target the study of scheduling algorithms. Generally they use interfaces that are specific to the simulator to specify an algorithm that is to be investigated.

GridNet [16, 17] is specialized on data replication strategies. Others, focused on network simulations are NS2 [18], OPNetModeler [19] and OMNet++ [20].

A general disadvantage of these simulators is that there are only few studies concerning their realism. Moreover, contrary to emulation, simulation requires to model

the environment *and* the application. This is convenient when the starting point is in fact a model of an application, namely an algorithm. It is not appropriate if the object under investigation is the implemented application itself.

2.3 Emulators

Microgrid [10] allows an execution of unmodified applications that are written for the Globus toolkit. Its main technique is to intercept major system calls such as `gethostbyname`, `bind`, `send`, `receive` of the application. Thereby the performance can be degraded to emulate a heterogeneous platform. This technique is invasive and limited to applications that are integrated into Globus. Because of the used round-robin scheduler, the measured resource utilization seems to be relatively inaccurate. Moreover and unfortunately, Microgrid is not maintained anymore.

eWAN [21] is a tool that is designed for the accurate emulation of high speed networks. It does not take CPU and memory capacities of the hosts into account and thus does not permit to perform benchmarks for an application as a whole.

ModelNet [11] is a tool principally designed to emulate the network component. It does not provide emulation of the CPU or memory capacities as we do in Wrekavoc. Moreover, it requires network emulator to be run on a FreeBSD machine where ours is a plain Linux solution.

Virtual machine (VM) technology is another approach that allows several guest to be executed on the same physical architecture. Moreover, CPU throttling implemented in VMs allows downgrading the performance of a node. However, as we will see below our solution is lighter and does not rely on such technology. Moreover, as far as we know there does not exists any environment based on VM that provides all the Wrekavoc features in a integrated way.

The *RAMP* (Research Accelerator for Multiple Processors) project [22] aims at emulating low level characteristics of an architecture (cache, memory bus, etc.) using field-programmable gate array (FPGA). Even if we show in this paper that we are already able to correctly emulate these features at the application level, such a project is complementary to this one and could be used to further improve the realism of Wrekavoc.

3 Wrekavoc

Wrekavoc addresses the problem of increasing the heterogeneity of a cluster in a controlled way. Our objective is to have a configurable environment that allows for reproducible experiments of real applications on large sets of configurations. This is achieved without emulating any of the code of the application but by degrading hardware characteristics. Four characteristics of a node are degraded: CPU speed, network bandwidth, network latency and memory size. Contrary to all other existing solutions described above, Wrekavoc allows for the simultaneous control of all these characteristics with an integrated and very simple mechanism.

Wrekavoc provides methodological support to scientists that need to perform experiments in the field of large-scale/heterogeneous computing. For instance, a typical

use case for Wrekavoc would be the study of a newly invented distributed algorithm for solving a computational problem on distributed environments. A test of the implementation of such an algorithm should not only run the designated program itself but also compare it with other existing solutions. A comparative benchmark should introduce as little experimental bias as possible and therefore simulation is generally not an option. However, large scale heterogeneous distributed environments that allow for well defined experimental conditions are not very common. Furthermore, they have a fixed topology and hardware setting, hence they do not cover a sufficiently large range of cases.

In such a situation, Wrekavoc enables us to take a homogeneous cluster (running under Linux) and to transform it into a multi-site heterogeneous environment by defining the topology, the interconnections characteristic, the CPU speed and the memory capacity. While only one homogeneous cluster is required, the possible configurations are numerous. Our only restriction is that every emulated node must correspond to a real node of the cluster. This then provides the desired platform to test and compare the designed program under a large range of different environment specifications.

3.1 Design goals

Wrekavoc was designed with the following goals in mind.

Transform a homogeneous cluster into a heterogeneous multi-site environment. This means that we want to be able to define and control the heterogeneity at a very low level (CPU, network, memory) as well as the topology of the interconnected nodes.

Ensure reproducibility. Reproducibility is a principal requirement for any scientific experiment. The same configuration with the same input must have the same behavior. Therefore, external disturbance must be reduced to the minimum or must be monitored so as to be incorporated into the experiment.

Provide simple commands and interfaces to control the heterogeneity.

Use software to degrade the performance of hardware Another possibility would consist in partially upgrading (or downgrading) the hardware (CPU, network, memory). By this, however, the heterogeneity is fixed and the possible control is very low. Hence, our approach consists in degrading the performance of the hardware by means of software as it ensures a higher flexibility and control of the heterogeneity.

Make the degradation of the features independent of each other. As we are going to degrade the different characteristics of a given node (CPU, network, memory), we want these degradations to be independent. For instance, we want to be able to degrade the CPU without degrading the bandwidth and *vice versa*.

Be realistic. We want Wrekavoc to provide a behavior as close as possible to the reality. Ensuring realism is necessary to assess the quality of the experiments and the confidence in the results.

3.2 Configuring and Controlling Nodes and Links

Wrekavoc uses a homogeneous physical network and builds an overlay network that has its proper characteristics in terms of topology, connectivity and performance. Wrekavoc's main notion to transform a homogeneous into an heterogeneous cluster is called an *islet*. An islet is *a set of nodes that share similar limitations*. Two islets can be linked together by a virtual network which can also be restricted as needed, see Fig. 1(b). Packets for islets that are not directly connected are routed through intermediate islets. If the islets are not connected, no communication is possible.

All islet configurations are stored in the first part of the configuration file, see Fig 1(a). In a second part of this file the network connection (bandwidth and latency) between each pair of islets is specified.

3.2.1 Specifying an Islet

Being a set of nodes that share similar characteristics, an islet is defined by several parameters. First we specify the number of nodes that are within this islet. Then, for each of these nodes we define the limitation parameters of each node (CPU, network, memory).

Theses parameters are randomized and can be specified in two ways. The value can follow a Gaussian distribution[1] of the form [*mean*;*std.dev.*] or it can follow a uniform distribution of the form [*min-max*]. For each islet, we define several parameters. `SEED` is an integer that is used for drawing distributed value according to the chosen random distributions. The special value of -1 indicates that the seed itself is drawn randomly for each run. `CPU` is a distributed value of the CPU frequency in MHz of the nodes of the islet. `BPOUT` (resp., `BPIN`) is a distributed value of the outgoing (resp., incoming) bandwidth in Mb/s. `LAT` is the distributed value of network latency in ms. `USER` is the the POSIX user ID for which the limitations are made. `MEM` is the distributed value of the usable memory in MiB.

Optionally, a gateway can be attached to an islet. A gateway is specified by its ID and by its input and output bandwidth. The role of a gateway is to emulate the contention between islets. The sum of the outgoing (resp., incoming) flow cannot exceed the OUT (resp., IN) capacity. If the sum of the flows requests more than the available bandwidth, then the gateway emulates a bottleneck and allocates a fair share of the available bandwidth to these flows.

3.2.2 Linking Islets Together

All islet configurations are stored in a configuration file. The network links between pairs of islets are described at the end of this file using the `!INTER` keyword (see Fig. 1(a) for an example). This keyword is followed by the two islets that are linked, then by the bandwidth distribution in each direction, then by the description of the latency between the two islets. The last number is the seed used for the random distribution.

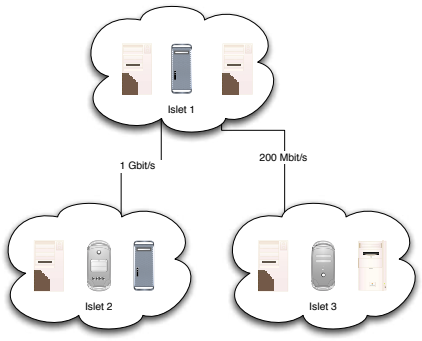
Note that if there is no gateway, Wrekavoc considers that there are as many links as pairs of machines between the islets, and hence, no congestion is emulated.

[1] each node has exactly the same value if we set 0 for the standard deviation.

3.2.3 Example

```
islet1 :  [20] {
   SEED: 1
   CPU : [1000;0]
   BPOUT : [100;0]
   BPIN : [100;0]
   LAT : [0;0]
   USER : user1
   MEM : [1024;0]
   GATEWAY : GW1
   ISLET_FORWARDING_CAPACITY_IN : 1000
   ISLET_FORWARDING_CAPACITY_OUT : 1000
}
islet2 :  [10] {
   SEED : -1
   CPU :[100-2000]
   BPOUT : [200;0]
   BPIN : [200;0]
   LAT : [10;0]
   USER : user1
   MEM :[512;0]
   GATEWAY : GW2
   ISLET_FORWARDING_CAPACITY_IN : 100(
   ISLET_FORWARDING_CAPACITY_OUT : 10(
}
islet3 :  [10] {
   SEED : -1
   CPU :[500;100]
   BPOUT : [200;0]
   BPIN : [200;0]
   LAT : [0.05;0]
   USER : user1
   MEM :[512;0]
   GATEWAY : GW3
   ISLET_FORWARDING_CAPACITY_IN : 200
   ISLET_FORWARDING_CAPACITY_OUT : 200
}
!INTER : [islet1;islet2] [1000;0] [1000;0] [10;0] 1
!INTER : [islet1;islet3] [200;0] [200;0] [100;0] 1
```

(a) Configuration file



(b) Logical view of islets

Figure 1: A configuration file and the corresponding logical view

Fig. 1(a) shows how to emulate 3 islets as shown in Fig. 1(b). Of course, this configuration has to be executed on a cluster having at least the required performance and the number of nodes (*i.e.*, 43: 40 for the islets and 3 for the gateways). In this example, *islet1*, is made of 20 nodes at 1 GHz with a 100 Mb/s interconnect with the minimum possible latency and 1 GiB of memory. *Islet2* comprises 10 heterogeneous nodes with frequency between 100 MHz and 2 GHz and a fast Ethernet network (bandwidth: 200 Mb/s, latency: 10 ms). The third islet is made of 10 nodes following a Gaussian distribution for the frequency with 500 MHz on average and a standard deviation 100 MHz.

The bandwidth is 200 Mb/s and the latency is 50 μs. Nodes of islet 2 and 3 have 512 MiB of memory.

Islet 1 and islet 2 are linked by a 1 Gb/s link with 10 ms latency. The backbone interconnecting islet 1 and islet 3 is asymmetric (200 Mb/s bandwidth from 1 to 3 and 100 Mb/s bandwidth from 3 to 1, with 100 ms of latency in both directions).

Islet 2 and islet 3 are not directly connected. This means that packets will be routed through islet 1.

Finally, gateways on each islet are identified by an alias. The inward and outward forwarding capacities are the input and output bandwidth of the gateways. For instance, the fact that the forwarding capacity is 1Gb/s for islet 1 means that if all the 20 nodes of islet 1 communicate with islet 2 and islet 3, then the aggregated bandwidth will not exceed 1Gb/s even if the sum of the 2 backbones is 1.2Gb/s (1 Gb/s bewteen islet 1 and islet 2 and 200 Mb/s bewteen islet 1 and islet 3).

Remark that using the value of -1 as seed for islet2 and islet3 means that each time we configure the nodes we will get a different configuration of the nodes. This allows the investigation of statistical properties over a large variety of configurations. Exact reproducibility of the node configuration is ensured by using positive seeds.

3.3 Implementation details

The implementation follows the client-server model. On each node for which we want to degrade the performance, a daemon runs and waits for orders from the client. The client is a controlling process that performs a specific experiment. It reads the configuration file and the list of available nodes we want to configure (the one where the daemons are running). Then, it sends configuration information that describes the heterogeneity settings to this daemon. When a server receives a configuration order, it degrades the node characteristics accordingly. The client can also order to recover the non-degraded state.

The layered architecture of our tool is depicted in Fig. 2. At the bottom level, we have the infrastructure we want to degrade and control. At the top level, we have each of the features of Wrekavoc: bandwidth and latency regulation, topology control through gateways, CPU degradation and memory limitation. The middle layer describes the technologies, tools and algorithms actually implemented in Wrekavoc to accomplish each features. We now detail each of these implementation.

3.3.1 CPU Degradation

We have implemented three different methods for degrading CPU performance. They have their particular advantages and drawbacks which we discuss in the sequel.

The first approach consists in managing the frequency of the CPU through the CPU-Freq interface of the Linux kernel. This interface was designed to limit the CPU frequency in order to save electrical power on laptops. It is based on proprietary CPU technologies such as AMD's *PowerNow!* or Intel's *SpeedStep* which are not always available on cluster nodes. Also, at most 10 different discrete frequency values are available through this interface.

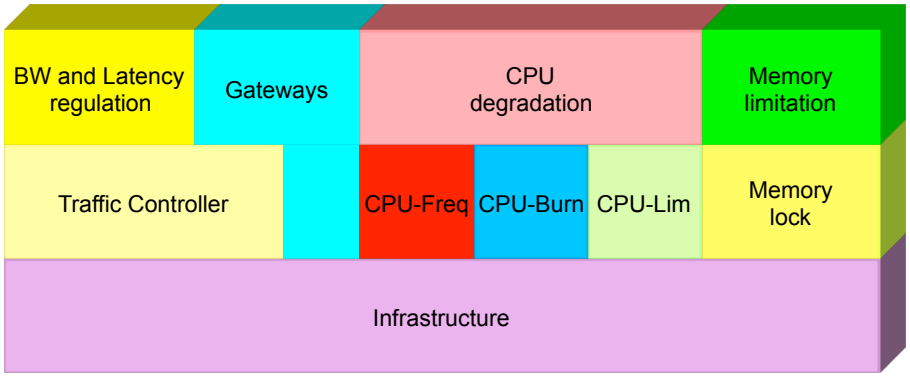


Figure 2: Wrekavoc layered architecture

The second approach is based on burning CPU cycles. A program that runs under real-time scheduling policy burns a constant portion of the CPU, whatever the number of currently running processes. More precisely, a CPU-burn sets the scheduler to a FIFO policy and gives itself the maximum priority. It then estimates the time it needs to make a small computation. This computation is blocking and therefore no other program can use the CPU. After the computation, the CPU burner sleeps for the corresponding amount of time and then iterates the whole process. A small tuning time is needed to make sure sleeping and calculation times are long enough in order to minimize time spent in system calls. The system call used to set the scheduler is `sched_setscheduler`. By means of the POSIX `sched_setaffinity` system call, each CPU burner is tied to a given processor on a multi-processor node. The main drawback of this approach is that the CPU limitation equally occurs for kernel and user mode processes. Therefore, as a result the network bandwidth may be limited by the same fraction as the CPU.

When an independent limitation of the CPU and the network is required, we propose a third alternative based on user-level process scheduling called CPU-Lim. A CPU limiter is a program that supervises processes of a given user. Using the `/proc` pseudo-filesystem, it accesses all relevant information about the processes that it has to limit (wall clock time, time passed in user or kernel mode, etc.). Based on that information, it suspends the processes when they have used more than the required fraction of the CPU using the `SIGSTOP` and `SIGCONT` signals, see Algorithm 1 for a formal description. This is the default method.

CPU-Burn and CPU-Lim have the side effect of keeping the CPU time of the process to be limited unchanged whatever the limitation imposed by Wrekavoc. Therefore, monitoring an application controlled by one of these methods should rely only on the wall clock time.

Algorithm 1: The CPU-Lim algorithm

```
Input: pid // The ID of the process to be limited
Input: percent // The percentage of limitation (between 0
     and 100)
sleeping=false // Is the process we monitor already
sleeping?
sleep_time=1 // How long we sleep after each loop
while process alive do
    // start_time:  date of the start of the process
    // utime:  time passed in user mode
    // stime:  time passed in kernel mode
    (start_time, utime, stime) = read_proc_stat(pid)
    // Compute wall time, cpu time and percentage of
    activity
    since_boot = read_proc_uptime()
    walltime = since_boot-start_time
    cpu_time = utime+stime
    cur_percent = cpu_time/wall_time
    // Based on the current activity of the process,
    sent it to sleep or awaken it
    if cur_percent > percent then
        if sleeping ≡ false then
            kill(pid,SIGSTOP)
            sleeping=true
    else
        if sleeping ≡ true then
            kill(pid,SIGCONT)
            sleeping=false
    // Sleep some time in order not to use too much CPU
    usleep (sleep_time)
    // As time passes, sleep longer as reactivity is no
    longer an issue
    if sleep_time < 100 then
        sleep_time++
```

3.3.2 Network regulation

Limiting latency and bandwidth is done using *tc* (traffic controller) [23]. The tool *tc* is based on *iproute2*, a program that allows advanced IP routing. With these tools, it is possible to control both incoming and outgoing traffic of a node. Furthermore, versions above 2.6.8 also allow for the control of the latency of the network interface. An important aspect of tc is that it can alter the traffic using numerous and complicated rules based on IP addresses, ports, etc. We use *tc* to define a network policy between each pair of nodes. This raises scalability issues as in a configuration with n nodes, each node has to implement $n - 1$ different rules. This issue will be discussed in the experimental section, see Sec. 4.1. Degradation of network latency and bandwidth is implemented using Class Based Queueing (CBQ): incoming or outgoing packets are stored into a queue according to the given quality of service before being transmitted to the TCP/IP stack. In order to function properly, a Linux kernel version above 2.6.8.1 is required and needs to be compiled with the `CONFIG_NET_SCH_NETEM=m` option.

To correctly emulate the bottleneck that can appear over a backbone link between two islets, we can dedicate a node that acts as a gateway for each islet. This gateway is responsible for forwarding TCP packets from one islet to another by sending these packets to the corresponding gateway. Gateways regulate bandwidth and latency using *tc* in the same way as regular nodes. This allows complex topologies between islets where some islets are directly connected and some are not. This is implemented by changing the local routing table of each node through the *ip* tool (of *iproute2* as well). Finally, for the case where an islet is not directly connected to another one, we have implemented a standard routing protocol[2] for forwarding packets between islets.

However, the use of gateways is not mandatory. Without specifying a gateway between two given islets, each pair of nodes communicates without an awareness of other communications that take place between different pairs. Up to the limits of the underlying physical network, the emulation works as if there were as many point-to-point links than there are pairs of processors between the islets. This allows for the modeling of platforms for which communication within islets is regulated while communication between islets is limited but does not suffer from contention. This covers the case where a cluster is able to communicate to another cluster without congestion.

Memory Limitation

In order to limit the total size of physical memory to a given target size, Wrekavoc uses the POSIX system calls `mlock` and `munlock` to pin physical pages to memory. These pages are then inaccessible to all applications and thus constrain the physical memory that is available to them.

Requirements

Based on the above description, we see that the requirements to install Wrekavoc on a cluster can be summarized as:

- *Linux kernel* 2.6.8.1 or newer.

[2] We use the RIP (Routing Information Protocol) that sets up routes by minimizing the number of hops

- *iproute2* with *tc* utility to control net traffic and *ip* utility to enable gateways.
- The *Gnu Scientific Library* to draw random numbers.
- The *XML 2 library* to build and parse configuration files.

4 Standalone validation

We have performed several experiments to assess the quality of Wrekavoc. The first series of experiments gives a look at features of Wrekavoc itself, in particular the configuration time for the startup of the daemons on the nodes and microbenchmarks for the particular characteristics of the architecture that Wrekavoc restricts.

4.1 Configuration time

The client reads the configuration file, parses the file and builds an XML file for each node. This XML file contains the necessary information to set up the nodes (*e.g.* CPU, Network and memory degradation, user ID of the processes to be limited, connections between islets, gateway, etc.). Then, this sub-configuration is sent to each node. When a node receives a configuration file, it configures its own characteristics according to this file.

Fig. 3 shows the configuration time against the number of nodes, *i.e.* the time it takes to configure all the nodes. 4 curves are shown as the number of nodes increases from 2 to 130: (1) all the nodes are in one islet, (2) half of the nodes are in one islet the rest in an other islet, (3) two nodes per islet and (4) one node per islet. The results show that the configuration time increases with the number of nodes. The worst case occurs when we have one node per islet (the same number of islets as nodes). Even in this case configuring 130 nodes takes only 22 seconds, while with two nodes per islets it takes less than 10 seconds. The configuration time increases quadratically with the number of islets. This must be so, because the XML file (sent over the network and parsed locally) contains all the connection between every islets, which are quadratic in number, and the configuration runs linearly in the size of the specification.

4.2 Micro-benchmarks.

Using micro-benchmarks, we have benchmarked each kind of degradation separately (CPU, latency and bandwidth).

4.2.1 CPU Benchmarks

To measure how performance degradation impacts the execution of a computation, we use the ratio between the expected and the actual duration times of a matrix multiplication benchmark. The expected time is computed by multiplying the the time without degradation with the specified degradation. A ratio of 1 means that the observed execution time matches the expected time. To perform this benchmark, we have run a sequential dense matrix multiplication found in GotoBlas 1.12 (`http:`

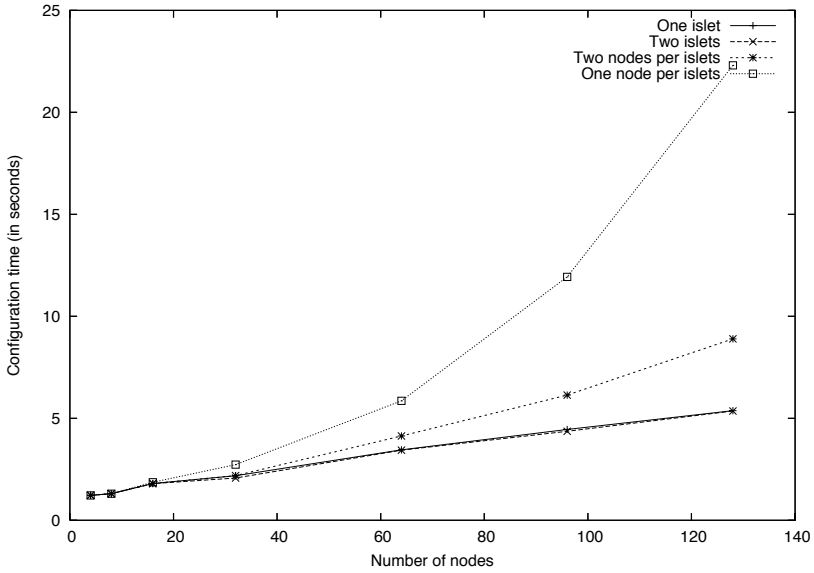


Figure 3: Configuration time for different islet sizes.

`//www.tacc.utexas.edu/tacc-projects`) 10 times. We chose this benchmark as it is a highly intensive computational kernel that directly measures the CPU power. On Fig. 4 we see that this ratio is never below 0.98, *i.e.* less than 2% of difference. Moreover we see that the normalized standard deviation is also very small. In conclusion, the CPU limitation behaves as expected with a small tendency to *over-do* the CPU downgrade: the ratio is lower than 1.

4.2.2 Network Bandwidth Benchmarks

Fig. 5 shows the observed bandwidth versus the desired bandwidth when one node sends a single message to an other node and while the rest of the network of idle. The size of the message varies between 10 kB 15 MiB. The physical network without degradation was a 1 Gb/s Ethernet network. Hence, points at 1000 Mb/s on the x-axis are real data, (*i.e.*, measured without regulating the network with Wrekavoc). The "*ideal*" line shows what one should obtain theoretically. The results shows that the obtained bandwidth is always very close to the desired one and hence we conclude that Wrekavoc is able to regulate the network at the desired value. We see that for 10 kB, we obtain a slightly greater bandwidth than the limited bandwidth. This is

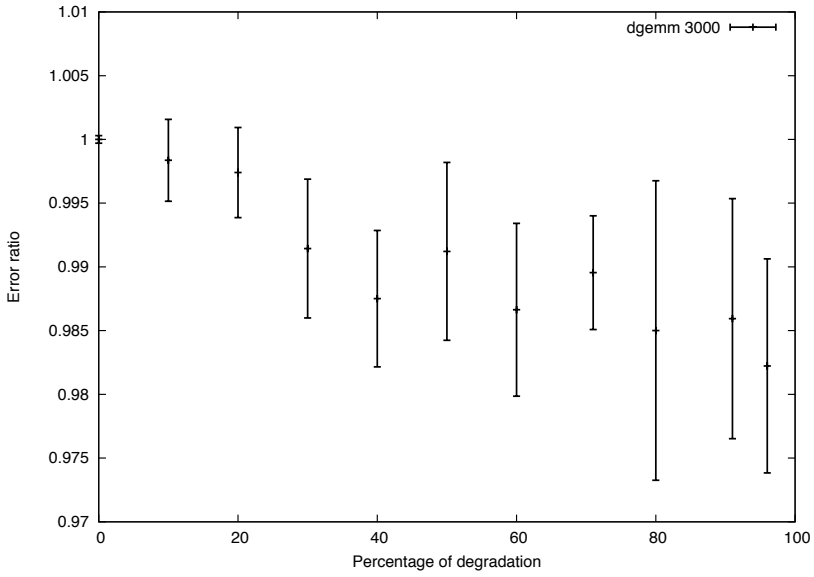


Figure 4: CPU micro-benchmark. Normalized average and standard deviation of 10 dgemm runs

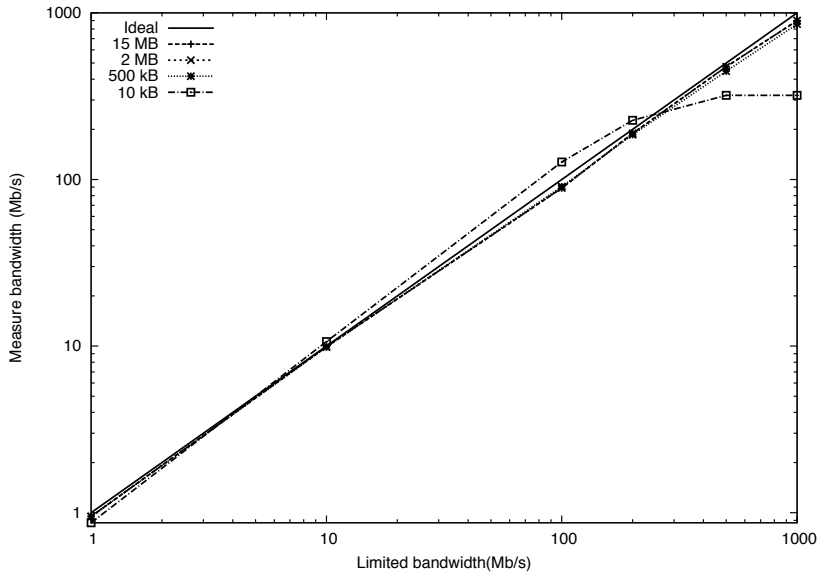


Figure 5: Bandwidth micro-benchmark.

due to the fact that *tc* uses some bucket to limit the bandwidth. Here, a bucket is a data structure of limited capacity that receives packets. The limitation starts when the bucket is completely filled. The amount of packets to fill the bucket being fixed, we see, for small messages that the real bandwidth is a little bit higher than the desired one. For this same size of data, we see that it is not possible to achieve the peak bandwidth. This phenomenon also shows in real network. Indeed, further investigations have shown that we obtain exactly the same bandwidth (320 Mb/s) for 1 Gb/s network card without network degradation for 10 kB messages by using two old PCs (PII at 400 MHz and PIII at 550 MHz) with a Gb PCI ethernet card under linux kernel 2.6.12 at runlevel 1.

4.2.3 Network Latency Benchmarks

Table 1 shows the average round-trip-time (RTT) obtained by the *ping* command with different degraded latencies. Results show that the RTT is very close to twice the value of the desired latency which is what one should expect as the latency is paid twice when doing a round trip.

set latency	1	5	10	50	100	500	1000
RTT	2.12	10.05	20.12	100.058	200.20	1000.05	1999.75

Table 1: Round-trip-time against desired latency in ms.

5 Validation through realistic applications

An experimental tool such as a simulator, an emulator or even a large-scale environment always provides an abstraction of reality: to some extent, experimental conditions are always synthetic. Therefore, the question of realism of the tools and the accuracy of the measurements is of extreme importance. Indeed, the confidence in the conclusions drawn from the experiments greatly depends on this realism. Hence, a good precision of the tools is mandatory to perform high quality experiments, to be able to compare with other results, for reproducibility, for calibration to a given environment, for possible extrapolation to larger settings than the given testbed, etc. However, a brief look at the literature shows that concerning simulators [8, 9] or emulators [10, 11], the validation of the proposed tools as a whole is seldom addressed.

Here, we validate the realism of Wrekavoc by comparing the behavior of the execution of a real application on a real heterogeneous environments and the same application using Wrekavoc. Such validation uses all the features of Wrekavoc (Network, CPU and memory degradation).

For each experiment, we build a set of configurations that match the real behavior. Due to the specificity of each type of the experiments (granularity, CPU usage, network usage, etc), these configurations vary slightly from one experiment to another. Being able to have a single Wrekavoc configuration for all the experiments is beyond what is currently achievable as we do not yet emulate low-level features such as cache or memory bandwidth.

ID	Proc	RAM (MiB)	System (Debian version)	Freq (MHz)	HDD type	HDD (GiB)	Network card (Mb/s)	MIPS
1	P. IV	256	2.6.18-4-686	1695	IDE	20	100	3393
2	P. IV	512	2.6.18-4-686	2794	IDE	40	1000	5590
3	P. IV	512	2.6.18-4-686	2794	IDE	40	1000	5590
4	P. III	512	2.6.18-4-686	864	IDE	12	100	1729
5	P. III	128	2.6.18-4-686	996	IDE	20	100	1995
6	P. III	1024	2.6.18-4-686	498	SCSI	8	1000	997
7	P. II	128	2.6.18-4-686	299	SCSI	4	1000	599
8	P. II	128	2.6.18-4-686	299	SCSI	4	100	599
9	P. II	128	2.6.18-4-686	298	SCSI	4	100	596
10	P. II	64	2.6.18-4-686	398	IDE	20	100	798
11	P. IV	512	2.6.18-4-686	2593	IDE	40	1000	5191
Front end	Dual Opt. 240	2048	2.6.18-4-amd64	1604	IDE	22	1000	3209

Table 2: Description of the heterogeneous reference environment

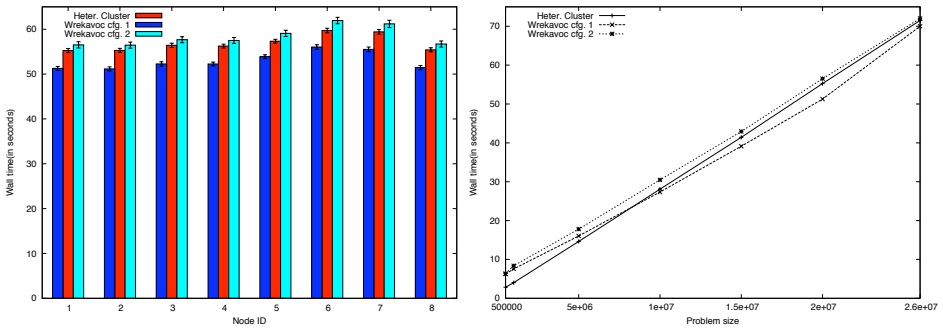
5.1 The heterogeneous reference platform

The heterogeneous platform we used as a reference was composed of 12 PC linked by a Gb switch. The characteristics of the nodes are described in Table 2. We have huge heterogeneity in terms of RAM, MIPS, clock frequency, network card, type and architecture of processors. All nodes have the same Linux distribution and kernel version and the same version of MPI, OpenMPI 1.2.2. MPI is necessary for executing some of the benchmark programs that used in the next sections.

Unfortunately, it has not been possible to perform experiments on a larger heterogeneous cluster. To the best of our knowledge, a heterogeneous environment with the desired characteristics (heterogeneity, scale, reproducible experimental conditions) that could be used to calibrate Wrekavoc against it is not available. For instance, experiments on Planet-lab are usually not reproducible and Grid'5000 is not heterogeneous enough.

The validation methodology employed here is the following. We have compared the execution of different applications on the heterogeneous cluster described above and on Grid'5000 clusters *heterogeneized* with Wrekavoc: an homogeneous cluster is transformed into an heterogeneous environment that closely reproduces the behavior of the reference cluster. The applications were chosen such that they cover a large variation of behavior: in particular the absence or the presence of different sorts of *load balacing algorithms* (static and dynamic with different schemes and paradigms). Moreover, to ensure reproducibility of our benchmarks we did not have any variance on the values in the configuration file. Last, to mimic the above platform, we have always used only one islet.

The benchmark programs that have been chosen are not themselves object of study, here, nor are they part of the Wrekavoc environment. Our goal is not to optimize or



(a) Times of each node of the heterogeneous cluster with 20,000,000 `doubles`.

(b) Times of machine "*ID 2*" with varying problem size.

Figure 6: Execution wall time for the sort application on the real and emulated heterogeneous cluster. Averages over 10 runs.

improve the behavior of these applications, but to show that Wrekavoc is well suited to predict this behavior. To be a valuable tool, Wrekavoc should be able predict problematic behavior, such as imbalance during an application run, and thus we must also benchmark programs that show such a behavior.

5.2 A fine-grain application without load-balancing

The first set of experiments we performed on the heterogeneous cluster aims at demonstrating the impact of a load imbalance. In this case, on the real environment, faster nodes will finish their computation earlier and will be idle during some part of the computation. We wanted to see how Wrekavoc is able to reproduce this behavior.

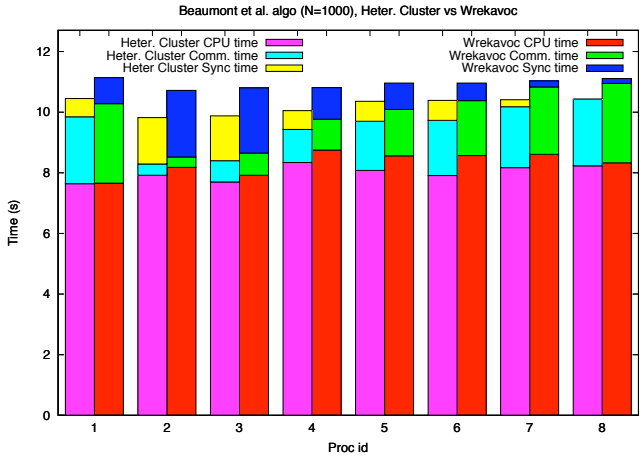
The application we used is a parallel sort algorithm implemented within the parXXL library [24]. The algorithm used is based on Gerbessiotis' and Valiant's sample sort algorithm [25].

We used 8 of the 11 nodes of the reference platform shown Table 2 (2,4,5,6,7,8,9,11): the processor type of node 1 was not supported by ParXXL, we chose only one of the identical nodes 2 and 3, and node 10 did not have enough memory.

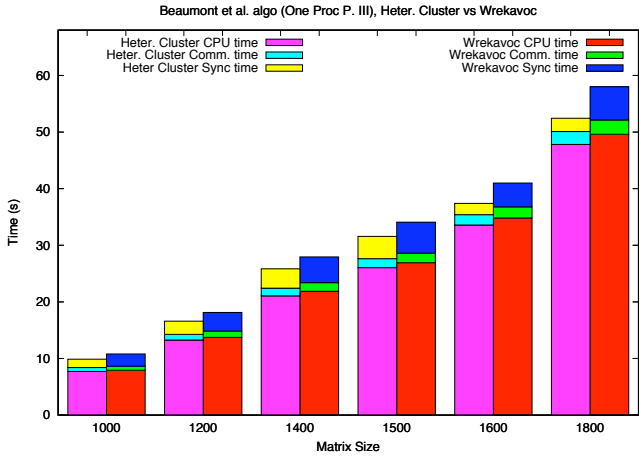
Fig. 6(a) shows the average wall time, for 10 executions, of each node for the heterogeneous cluster and two wrekavoc configurations.

In order to see the difference, we performed 5 sorts of 20,000,000 `doubles` (64 bits) in a row. Results show that Wrekavoc is able to reproduce the reality with a good accuracy as both configurations are close upper (or under) approximations. In the worst case, the Wrekavoc configuration results have a difference of 8% from the real platform.

In Fig. 6(b), we show the average wall time of 10 runs for the first node when varying the problem size from 500,000 to 20,000,000 doubles. We see that Wrekavoc has some difficulties in correctly emulating the reality for small size problem. However,

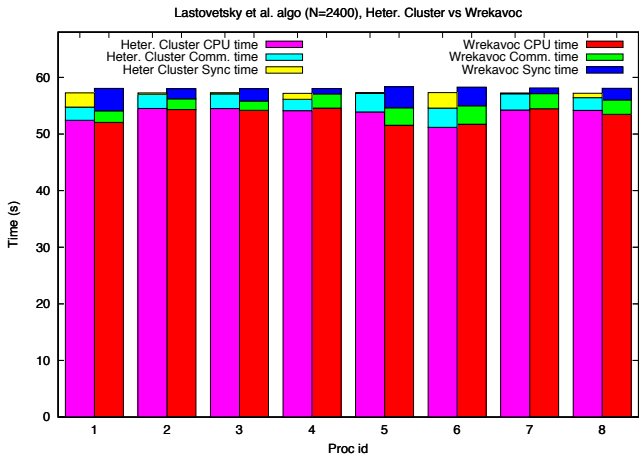


(a) Beaumont *et al.* algorithm, $N = 1000$

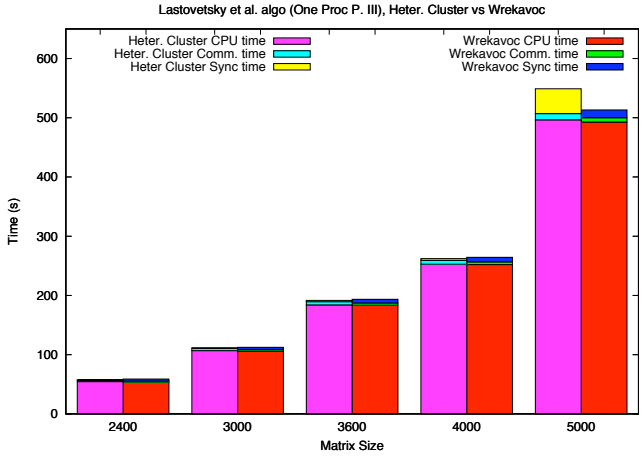


(b) Beaumont *et al.* algorithm for the same processor

Figure 7: Comparison of node runtime (CPU, communication and synchronization) for the static load balancing application (matrix multiplication – Beaumont *et al.* algorithm). On each figure, the bars on the right are for Wrekavoc and the bars on the left are for the heterogeneous cluster.

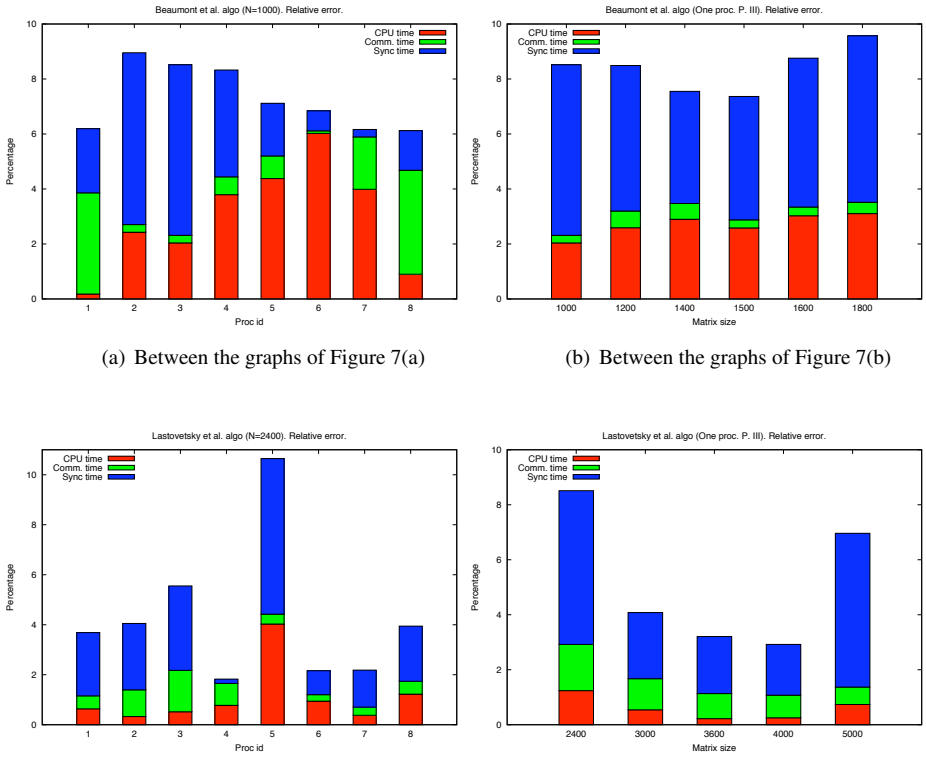


(a) Lastovetsky *et al.* algorithm, N=2400



(b) Lastovetsky *et al.* algorithm for the same processor

Figure 8: Comparison of node runtime (CPU, communication and synchronization) for the static load balancing application (matrix multiplication – Lastovetsky *et al/* algorithm). On each figure, the bars on the right are for Wrekavoc and the bars on the left are for the heterogeneous cluster.



(a) Between the graphs of Figure 7(a)

(b) Between the graphs of Figure 7(b)

(c) Between the graphs of Figure 8(a)

(d) Between the graphs of Figure 8(b)

Figure 9: Relative errors. Contribution of each of the timings (CPU, communication and synchronization.)

for large sizes, when the running time is above 20 seconds, the estimation is very realistic with an error margin below 10%.

5.3 Static load balancing

Here, we have benchmarked applications that perform a static load balancing at the beginning of the application according to the speed of the processors and the amount of work to perform. We have implemented two algorithms that perform parallel matrix multiplication on heterogeneous environments. The first algorithm from Beaumont *et al.* [26] is based on a geometric partition of the columns on the processors. The second from Lastovetsky *et al.* [27] uses a data partitioning based on a performance model of the environment.

We monitored both programs at the MPI level to measure CPU, synchronization and communication time. We use synchronous send and receive call and timed these

calls to measure the communication time. Hence, the communication time is not overlapped by computation. Moreover, we have added an MPI barrier to synchronize processes and measure the time each of them spend in the barrier, *i.e.* neither communicating nor computing.

We used 8 of the 11 nodes of the reference platform shown Table 2 (1,4,5,6,7,8,9,10): The other nodes were more powerful than the homogeneous Grid'5000 cluster used for this experiment and hence it was not possible to emulate them.

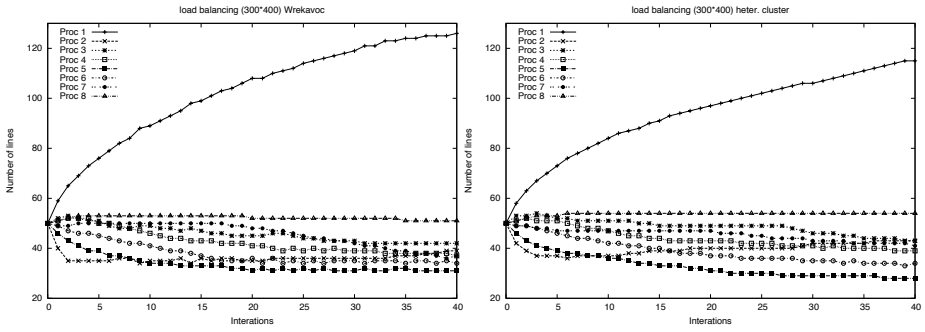
Experiments performed on the heterogeneous cluster where highly reproducible and hence the plot are the average of 3 measures. Concerning the homogeneous cluster with Wrekavoc plots are the average of 10 measures.

In Fig. 7(a), we show the comparison between the CPU, communication and synchronization times for the Beaumont *et al.* algorithm for matrix sizes of 1000. Nodes are sorted by CPU time. We see that the Wrekavoc behavior is very close to the behavior when using the heterogeneous cluster. Also, the proportions of the timings (CPU, communication and synchronization) are preserved. In order to discuss the differences between the two graphs of Fig. 7(a) quantitatively, we plot the relative error in Fig. 9(a). More precisely, in this graph we show the sum of the relative errors of all the timings (CPU, communication and synchronization) of the graphs of Fig. 7(a): for each x value i and each timing (CPU, communication and synchronization), we compute the relative error $100 \times \text{abs}(c_h(i) - c_w(i))/C_h(i)$ where $c_h(i)$ (resp., $c_w(i)$) is the value of the timing for the heterogeneous (resp., Wrekavoc) case for node i and $C_h(i)$ is the sum of the value of the timings on the heterogeneous cluster for node i. In the figure, the different colors then represent the contributions of the three resources.

We see that the overall relative error is always below 10%. The worst case is for processor 5 for which the error is mainly due to the synchronization time. In absolute values, this timing is very small (less than 0.6 seconds).

In Fig. 7(b), we show the comparison between the CPU, communication and synchronization time for the Beaumont *et al.* algorithm for varying matrix size on a fixed node. We see that the Wrekavoc behavior is very close to the behavior when using the heterogeneous cluster. The only problem concerns an increasing shift of the timings when matrix size increases. Indeed, in Fig 9(b), we stack the contribution of the relative error between Wrekavoc and the heterogeneous cluster of all the timings (CPU, communication and synchronization) of the graphs of Fig. 7(b). We see that the overall relative error is always lower than 10%. Moreover, we see that the contribution of communication to the error is marginal (less than 0.6 %). This means that, here, Wrekavoc was able to emulate communication with a great precision.

In Fig. 8(a) and 8(b), we present the same measurements as in Fig. 7(a) and 7(b) but for the Lastovetsky *et al.* algorithm. Here again, we see that Wrekavoc is very realistic with, again, a small shift in execution time when the matrix size increases. In Fig 9(c), we stack the contribution of the relative error between Wrekavoc and the heterogeneous cluster of all the timings (CPU, communication and synchronization) of the graphs of Fig. 8(a). From this figure, we can see that the relative error is very low in general (lower than 6%). There is one exception for processor 5 where the error is a little bit larger than 10%. The relative error of the graphs of Fig. 8(b) are shown in Fig. 9(d). Results show that these errors are always lower than 8% and that the CPU



(a) emulation with Wrekavoc (b) heterogeneous cluster

Figure 10: Comparison of the evolution of the load-balancing for executing the advection diffusion application

and communication times do not contribute a lot to these errors showing that Wrekavoc is doing a good job for emulating the heterogeneous hardware.

In conclusion to this section, we see that Wrekavoc precisely emulates the heterogeneous cluster (within 10%) in the general case. Moreover, most of the relative error is caused by synchronization. Last, while both algorithms are different and hence provide different raw performance, Wrekavoc is able, in both cases, to match their behavior.

5.4 Dynamic load balancing for iterative computation

Here, the dynamic load balancing strategy consists in exchanging some workload at execution time in function of the progress in the previous iteration. The program we have used solves an advection-diffusion problem (kinetic chemistry) described in [28]. Here, the load corresponds to the number of matrix lines held by each process. At each iteration to balance the load the nodes send or receive some rows to their neighbours.

As for the previous experiments, we used 8 of the 11 nodes of the reference platform shown Table 2 (1,4,5,6,7,8,9,10): The other nodes were more powerful than the homogeneous cluster we used here and hence it was not possible to emulate them.

In Fig. 10, we show the evolution of the load balancing of this application. At each iteration, we have monitored the number of matrix rows held by each processor. We plot the average number of 5 executions during the whole execution of the application for a problem on a surface of 300 columns and 400 matrix rows. There are 40 iterations. The results show that the evolution of the load balancing using Wrekavoc (right) or the heterogeneous cluster (left) are extremely similar. Processor 1 (the fastest), holds an increasing amount of load in both situation. More interestingly, processor 2 starts off with a very low load and then its load increases. Moreover, processor 5, the slowest one, is the least loaded at the end.

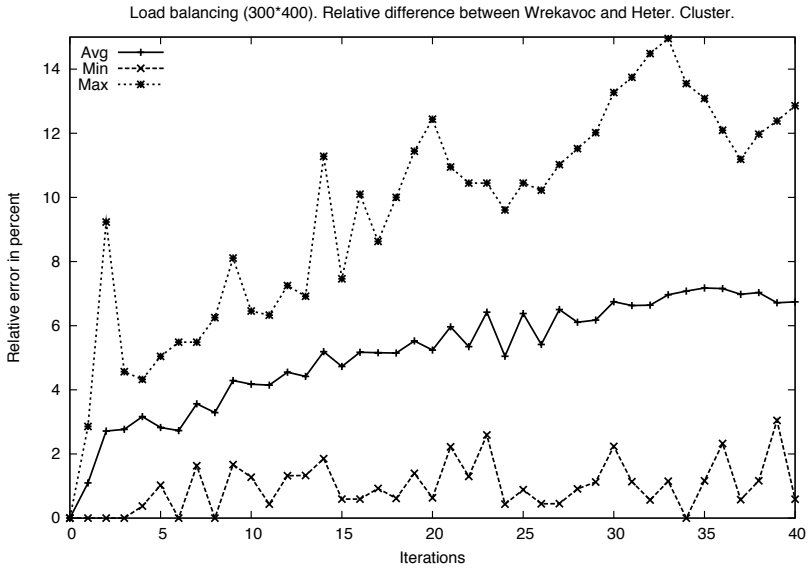


Figure 11: Relative error in percent between graphs of Figure 10

In Fig. 11, we show the relative error between the two graphs of Fig. 10. More precisely, for the eight processors involved in these experiments we computed the relative error as:

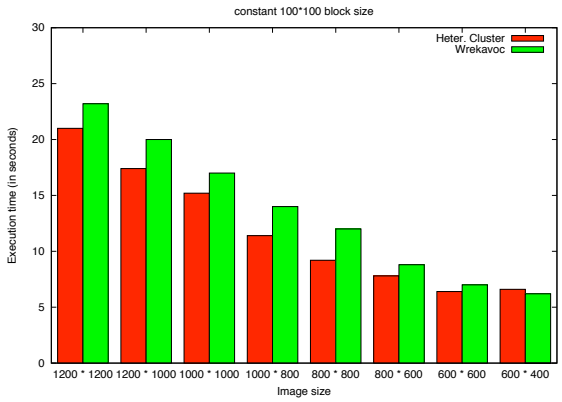
$$100 \times \text{abs}(n_w(i) - n_h(i))/n_h(i) \text{ for } i \in [1, 40]$$

(where $n_h(i)$ (resp., $n_w(i)$) is the number of lines treated by a processor of the heterogeneous cluster (resp., Wrekavoc case) at iteration i. Then, for each iteration, we plot the maximum, minimum and average value. We see that the error never exceeds 15% on maximum and 7% on average. Moreover the average error relatively stable after 30 iterations.

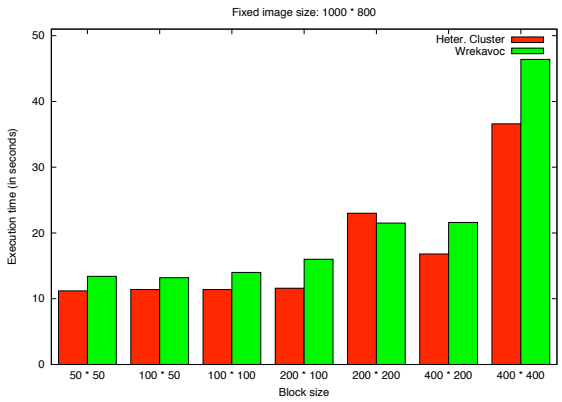
In summary, the predictability of the behavior of the emulated application is very high.

5.5 The master-worker paradigm

The next set of experiments concerns the master-worker paradigm. We have a master that holds some data and a set of workers that are able to process the data. When a worker is idle, it asks the master for some data to process, performs the computation and sends the results back. Such paradigm allows for a dynamic load balancing. As an application of this paradigm, we have chosen parallel image rendering with the



(a) block size of 100×100 and different image sizes



(b) image size 1000×800 with different block sizes

Figure 12: Parallel rendering times of the master-worker application

Povray [29] ray-tracer. Here, the master holds a synthetic description of a scene. This scene is decomposed into blocks such that each part can be processed by a worker.

We used 6 of the 11 nodes of the reference platform shown Table 2 (1,2,4,5,6,7): node 3 is equivalent to node 2 and hence was not used, node 8 and 9 had performance too close to node 7, node 10 was not powerful enough fort Povray and node 11 was too powerful to be emulated on the homogeneous Grid'5000 cluster.

In Fig. 12(a) (resp., 12(b)), we present the comparison between the running time of the application when the block size is fixed (resp., the image size is fixed) and the image size varies (resp., the block size varies). Each plot is the average of 5 runs.

We see that, most of the time, Wrekavoc is able to match reality, at least quantitatively. Indeed, if we order the block-size by running time the Wrekavoc order is the same than the heterogeneous cluster order.

5.6 The work stealing paradigm

Here, we benchmark the behavior of different work-stealing algorithms. Within this paradigm, an underloaded node chooses another node to ask for some work. Our chosen application for this methodology is the N-queen problem. This problem consists in placing N queens on a check board of size $N \times N$, such that no queen is blocked by any other one. The goal of the application is to find all the possible solutions for a given N. To solve this problem, we place M queens on the top M rows and generate, for each possible placement (there exists N^M such placement), a sub-problem that has to be processed sequentially. This problem is irregular: two sub-problem restricted to the same number of rows can have very different number of solutions and may require very different running times.

When a node is underloaded, different strategies can be applied to choose the node where work is stolen.

Algo1: c1	d1	g1 (random load stealing, load evenly distributed, granularity 14)			
Algo2: c2	d1	g2	**Algo3**: c1	d2	g2
Algo4: c1	d3	g2	**Algo5**: c1	d4	g2
Algo6: c1	d5	g2	**Algo7**: c1	d1	g5
Algo8: c1	d1	g3	**Algo9**: c1	d1	g1

Strategy c1: choose node at random; **c2**: chose one of two designated neighbors (nodes are arranged according to a virtual ring). The way the load is distributed initially may also have an impact. Here, we use 5 different strategies. **Distribution d1**: The load is evenly distributed; **d2**: Place all the load on the fastest node; **d3**: Place all the load on the slowest node; **d4**: Place all the load on an average speed node; **d5**: Place all the load on the second slowest node. When solving the N-queen problem, the load is composed of tasks that have a given granularity (*i.e.*, M, the number queens already placed). We fixed $N = 17$ and studied different granularities. **Granularity g1:** 2 rows; **g2:** 3 rows; **g3:** 4 rows; **g4:** 5 rows.

With these different characteristics, we have chosen 9 different algorithms out of the 50 possible combinations.[3] These 9 algorithms are depicted in the above table.

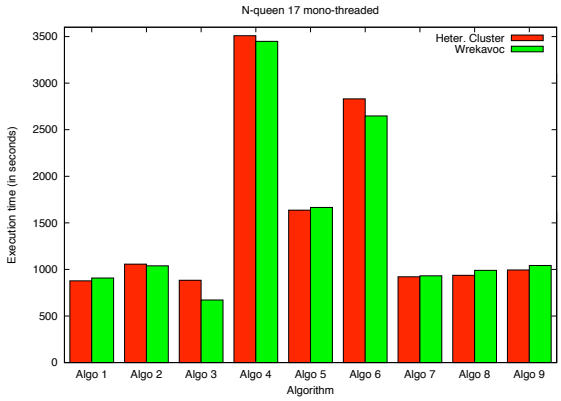


Figure 13: Computation time of the different algorithms for the N-queen problem of size 17, using the heterogeneous cluster or Wrekavoc, kernel 2.6.18

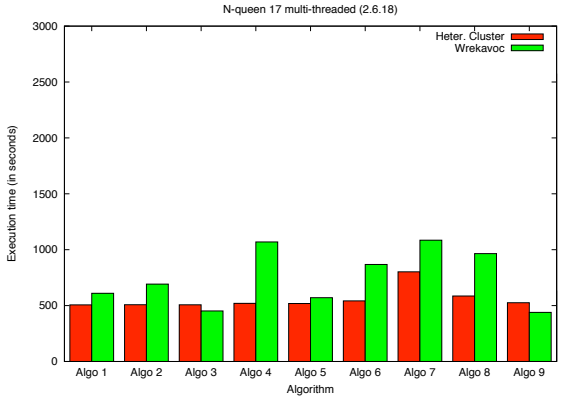
We used 8 of the 11 nodes of the reference platform shown Table 2 (1,4,5,6,7,8,9,10): The other nodes were more powerful than the homogeneous Grid'5000 cluster used for this experiment and hence it was not possible to emulate them.

The results are average timing of 6 runs. In Fig. 13, we present the results for the single-threaded implementation of the 9 algorithms. We see that Wrekavoc is able to reproduce the behavior of the heterogeneous cluster precisely. Timings are almost the same in both situations.

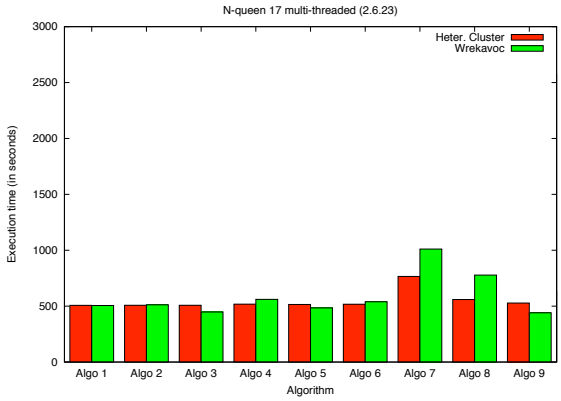
We have implemented a multi-threaded version of the algorithm; one thread for the computation and one thread for the communication. We have benchmarked Wrekavoc with two kernel versions 2.6.18 and 2.6.23. The difference between these two versions is a change in the process scheduler. The 2.6.18 version uses the $O(1)$ *scheduler*, while the 2.6.23 version of the kernel implements the so-called *Completely Fair Scheduling* (CFS), based on *fair queuing* [31]. Both schedulers are kernel developments of Ingo Molnár from Redhat Corp.

We present the results for both kernel versions in Fig. 14. We first see that the multi-threaded version is faster than the single-threaded (especially when the load is initially placed on slow processors (Algorithms 4, 5 and 6)). We see that Wrekavoc is able to reproduce this acceleration from the single threaded version to the multi-threaded. Moreover, we see that using kernel 2.6.23 improves the accuracy of Wrekavoc when experimenting a multi-threaded program. This holds especially for algorithms using the g2 granularity (algorithms 1 to 6). A higher granularity reduces the realism because

[3] We favored Strategy c1 since it is known to be the best for this application [30]



(a) With kernel 2.6.18



(b) With kernel 2.6.23

Figure 14: Computation time of the different algorithms for the N-queen problem of size 17, using the heterogeneous cluster or Wrekavoc, multi-threaded version, kernel 2.6.18 and 2.6.23

the number of tasks to execute lowers when the granularity increases and hence makes it difficult to reproduce the behavior.

We explain the fact that the realism of Wrekavoc is better with kernel 2.6.23 than with kernel 2.6.18 by the difference of the process scheduler implemented in this two versions. In Unix processes are scheduled according to their priority. This priority changes with the behavior of the process. The problem for the $O(1)$ scheduling algorithm of kernel 2.6.18 is that Wrekavoc adds a bias in the way the process priority is computed by stopping and continuing process on for its own purpose. Indeed, the $O(1)$ scheduler tends to favor I/O restricted processes. When Wrekavoc suspends a process for controlling the CPU speed and wakes it up later, the scheduler increases the process priority (taking it for a I/O bound process). Therefore, both threads (communication and CPU) of the process have acquired the same priority whereas the communication thread should have a higher priority: the process does not spend enough time in communication. Therefore, the behavior is extremely unrealistic when all the load is on a slow processor at the beginning. The CFS scheduling algorithm (of kernel 2.6.23) solves this problem giving extra priority to suspended processes similar to those that had been scheduled off when waiting for I/O. In conclusion, this shows that emulation of multi-threaded programs is realistic if running Wrekavoc on a recent Linux kernel version (at least 2.6.23).

6 Scalability

In the previous section, we could not use more than 12 processors because it was not possible to have a larger heterogeneous reference cluster under satisfying experimental conditions. Nevertheless, we want to evaluate the scalability of Wrekavoc on a large setting using hundreds of nodes. Since there is no real execution to compare it with, we will only be able to analyze the behavior on the environment heterogeneized by Wrekavoc.

6.1 CPU intensive benchmarks

Here, we use the Lastovetsky et al. matrix multiplication algorithm for heterogeneous environment. As in the previous experiment, we used the Grelon Cluster of the Nancy Grid'5000 site.

We have used different configurations. Each configuration is made of 8 islets. Each islet has one gateway to communicate with the other 7 islets. We have two settings of CPU speeds and two settings of network regulation. Concerning CPU speeds we have:

- **Fast CPU speeds**: all nodes of islet 1: 1550 MHz, islet 2: 1400 MHz, ... islet 8: 500 MHz.
- **Slow CPU speeds**: all nodes of islet 1: 775 MHz, islet 2: 700 MHz, ... islet 8: 250 MHz.

Concerning Network speed, we have:

- **Fast Net speeds** A clique at 1Gb/s and min latency (the latency of the underlying cluster without degradation).
- **Slow Net speeds** A clique at 100 Mb/s and 0.05 ms latency.

By combining network and CPU speeds, we obtain 4 different settings. For each setting, we use a different number of nodes per islet (from 1 to 13) leading to configuration having between 8 to 104 nodes.

In Fig. 15, we show the wall time of the matrix multiplication algorithm on each setting and for different number of nodes when the matrix size increases. Each plot is the average of at least 5 runs.

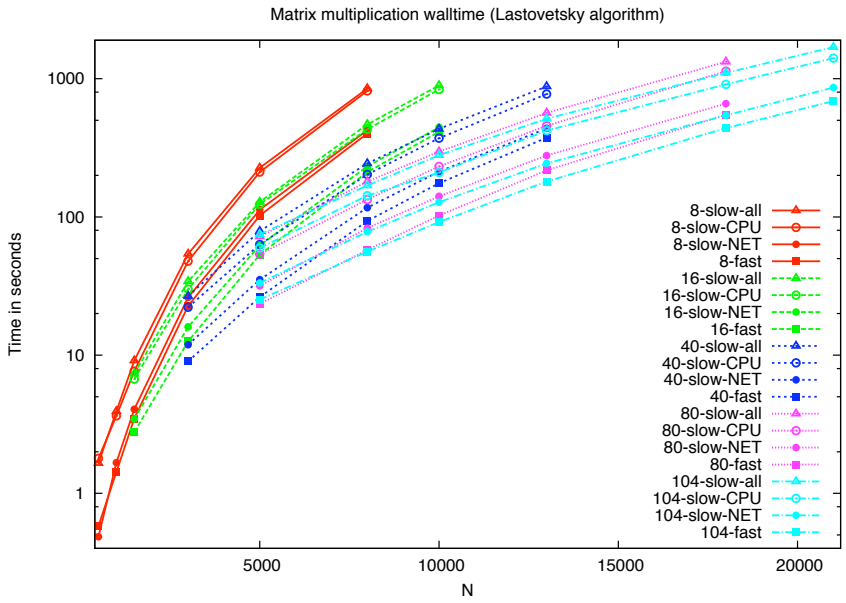
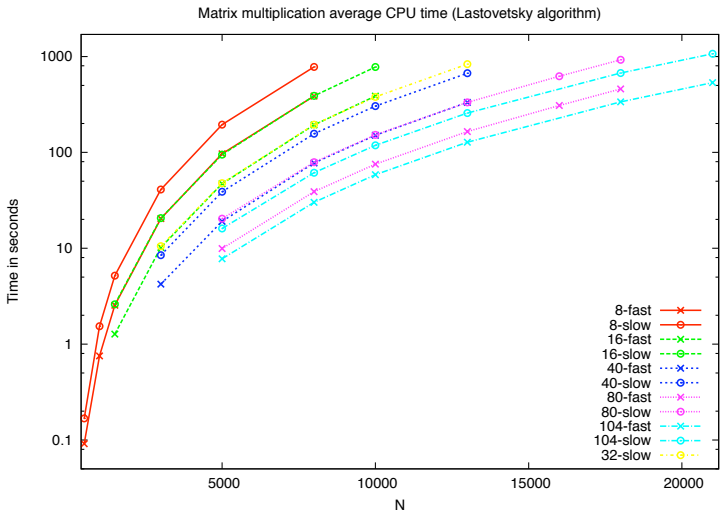
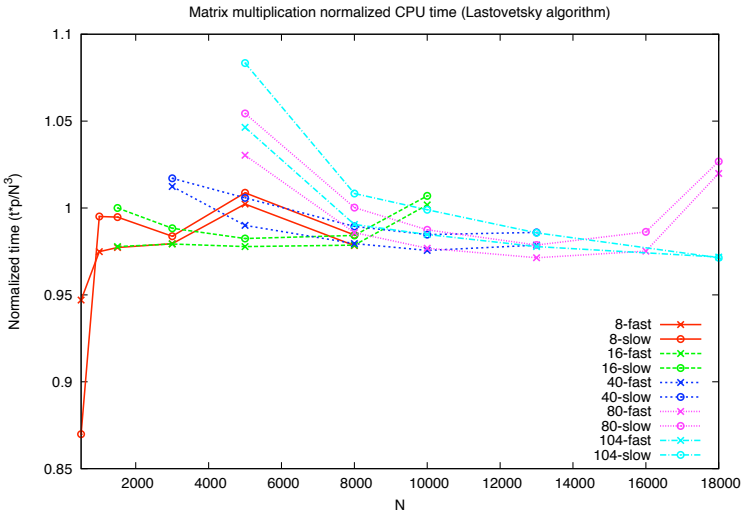


Figure 15: Average wall time of the matrix multiplication on different settings, different number of nodes and different matrix size

Results show that the behaviour is quite realistic. First, as the y-axis is in log-scale the shape of the plot follows a logarithmic shape which is what is expected. Second, we see that for a given number of nodes, the execution wall time is always ordered according to the ranking of the settings: slow network and slow CPU (the triangle dots), then fast network and slow CPU (the circle dots), then slow network and fast CPU (the plain circle) and lastly fast network and fast CPU (plain square). We also see that the network speed is of low importance concerning the whole wall time. Therefore, we analyze the CPU time of all the nodes (*i.e.*, the wall time minus the communication time and synchronization time). In Fig. 16(a), we plot the average cputime of all the nodes.

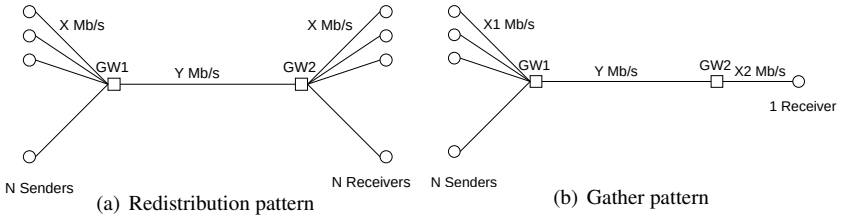


(a) CPU time



(b) Normalized CPU time

Figure 16: Average cputime and normalized of the matrix multiplication on different settings different number of nodes and different matrix size



(a) Redistribution pattern (b) Gather pattern

Figure 17: Communication pattern for benchmarking gateway scalability. X is the bandwidth of the intra-islet communication and Y of the bandwidth of the inter-islet communication

We observe another very interesting property. When we use a fast setting (cross) with a given number of nodes, the emulated CPU time is exactly the same with twice as much processors but for a slow setting (circle dots). This is exactly what we would expect with a perfect load balancing: islets in the fast setting are made of nodes that are twice as fast than in the slow setting. This shows that the Lastovetsky algorithm is able to perform a very good load balancing and that Wrekavoc emulates the CPU speed with a great precision. This also indicates that it is possible to normalize all the CPU time according to the number of nodes, the CPU setting, and a given reference point. Actually, based on the fact that the number of operations of a matrix multiplication is N^3 where N is the order of the matrix, we have the following normalizing formula:

$$\frac{t \times p \times f}{N^3}$$

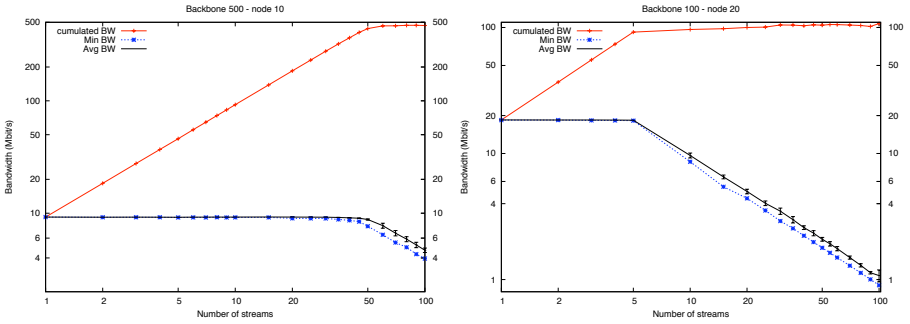
where t is the running time, p is the number of used processors and f equals 1 slow CPU setting and 2 for fast CPU setting.

In Fig. 16(b), we plot the normalized cputime according to the above formula. We have further normalized the plot such that the slow setting with 16 nodes and $N = 2000$ point has a value of 1. Based on that figure we see that all the points are within 10 percent of the expected value. The only exception is for slow setting, 8 nodes, $N = 5000$ for which the error is 13%. This is due to the fact that for that particular setting, the process durations are very small (in the order of 10 ms).

6.2 Scalability of gateways

In order to benchmark the gateways and their scalability, we have set two configurations. Such configurations lead to an overlay network whose topology is depicted in Fig. 17(a) for a redistribution pattern and Fig. 17(b) for a gather pattern.

To perform the benchmark we used the GDX Cluster of the Orsay Grid'5000 site. These benchmarks consists in measuring the bandwidth that is observed for network streams between two nodes of different islets. The expected behavior is that the aggregated bandwidth should increase linearly until it reaches the bandwidth of the inter-islet link. Beyond that point, the inter-islet bandwidth is shared between the different com-



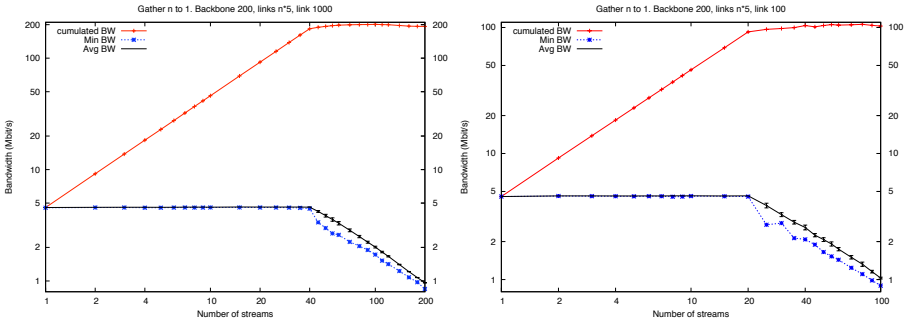
(a) Backbone: 500 Mb/s; inter-islet: 10 Mb/s (b) Backbone: 100 Mb/s; inter-islet: 20 Mb/s

Figure 18: Accumulated bandwidth when varying the number of senders (streams) between the two islets for the redistribution pattern

municating streams. Thanks to TCP this sharing must be fair. This means that the amount of allocated bandwidth must roughly be the same for every stream.

In Fig. 18, we plot the accumulated bandwidth for two different redistribution patterns inter-islet at X=10 Mb/s and intra-islet at Y=500 Mb/s or X=20 Mb/s and Y=100 Mb/s respectively, when increasing the number of senders. Each point is the average of at least four measures. Results show that the behavior of Wrekavoc is very realistic and always within 10% of what is expected. For instance, for the left graph, the bandwidth accumulates up to 50 streams where it reaches the backbone bandwidth and then stabilizes between 450 and 500 Mb/s. Moreover, we see that the average bandwidth of each stream is stable up to the point where contention starts. As the error bars (std. dev.) are very small, we deduce that each stream has roughly the same bandwidth. Finally, the minimum bandwidth curve shows that, as expected with TCP, each stream is allocated a minimum amount of the total bandwidth ensuring a good quality of service.

In Fig. 19, we plot the accumulated bandwidth when each sender emits a stream at X1=5 Mb/s, a backbone Y=200 Mb/s, and the receiver is connected to the gateway at X2=1 Gb/s or X2=100 Mb/s. For X2=1 Gb/s the backbone is the bottleneck and result show that the accumulated bandwidth increases up to 40 streams and then stabilizes between 184 and 200 Mb/s. Here again, we see that the results deviate by at most 10% from the prediction. For X2=100 Mb/s, the receiver is the bottleneck: the accumulated bandwidth increases up to 20 streams and then stabilized around 100 Mb/s with an error less than 10%. We can also make the same conclusions as for the redistribution case, concerning the stability of the bandwidth for each stream up to the contention point, the roughly equal share of bandwidth given to each stream and the good quality of service that is ensured by TCP is still achieved with Wrekavoc.



(a) X1=5 Mb/s, Y=200 Mb/s and X2=1 Gb/s (b) X1=5 Mb/s, Y=200 Mb/s and X2=100 Mb/s

Figure 19: Accumulated bandwidth when varying the number of senders (streams) between the two islets for the gather pattern

7 Conclusion and Future Work

Nowadays computing environments are more and more complex. Analytic validation of solutions for these environments are not always possible or not always sufficient.

In this work, we propose a new approach called Wrekavoc for emulating heterogeneous distributed computing platforms that are based upon the Linux kernel. Wrekavoc defines and controls the heterogeneity of a given platform by degrading CPU, by regulating the network or by limiting the memory of each node and by composing the emulated platform as a static overlay. The overall goal is to provide an environment on which reproducible experiments on heterogeneous settings are eased.

Our current implementation of Wrekavoc has been tested, benchmarked and evaluated in several ways. First, we have shown that configuring a set of nodes is very fast. Micro-benchmarks show that we are able to independently degrade CPU, bandwidth and latency to the desired values.

Second, we have validated Wrekavoc by comparing the execution of several real applications on a real environments to a reference cluster running Wrekavoc. The results obtained in the experiments concern all the features of Wrekavoc (network regulation, memory limitation and CPU degradation). Wrekavoc is realistic and has a very good reproducibility. Moreover, the tool provides emulations that are not tied to the real host platform: at the application/user level, different architectural features (*e.g.*, processor architecture, memory bandwidth, cache, instruction sets, etc.) are correctly emulated.

Last, we have performed a set of experiments to assess the scalability of Wrekavoc. Concerning scalability at the CPU level we have performed benchmarks using more than 100 nodes and for those concerning the network up to 200 nodes. In both cases, we see that the behaviour is very close to what is expected.

All these results show that Wrekavoc is a suitable tool for developing, studying and comparing algorithms in heterogeneous environments: in almost all cases results are

within 10% of what is expected (from theory or other real benchmarks). The realism is even better when the runtime of the program under investigation or the communication time is long enough (1s or more).

Future work are directed to the improvement of the realism of Wrekavoc for low-level features such as cache, memory bandwidth or disk IO. Moreover, it seemed important to us to first be able to emulate static heterogeneous platforms accurately. In a second step, we now hope to be able to emulate dynamic features. Hence, future work will be directed towards the ease of the calibration of the environment, a better emulation of multi-threaded programs and to new modeling features such as node volatility and dynamic load.

Acknowledgment

This work was supported by a one year internship grant for the second author by FONGECIF to INRIA Nancy – Grand Est.

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see `https://www.grid5000.fr`).

References

[1] I. Foster, C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, Morgan-Kaufmann, 1998.

[2] The Grid'5000 experimental testbed.
URL `https://www.grid5000.fr`

[3] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for internet applications, in: SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, ACM, New York, NY, USA, 2001, pp. 149–160. `doi:http://doi.acm.org/10.1145/383059.383071`.

[4] A. Oram (Ed.), Peer-to-Peer: Harnessing the Power of Disruptive Technologies, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[5] D. P. Anderson, Boinc: A system for public-resource computing and storage, in: GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, IEEE Computer Society, Washington, DC, USA, 2004, pp. 4–10. `doi:http://dx.doi.org/10.1109/GRID.2004.14`.

[6] G. Fedak, C. Germain, V. Néri, F. Cappello, Xtremweb: A generic global computing system, in: CCGRID, IEEE Computer Society, Los Alamitos, CA, USA, 2001, pp. 582–587.

[7] J. Gustedt, E. Jeannot, M. Quinson, Experimental Methodologies for Large-Scale Systems: a Survey, Parallel Processing Letters 19 (3) (2009) 399–418.

[8] H. Casanova, A. Legrand, M. Quinson, SimGrid: a Generic Framework for Large-Scale Distributed Experiments, in: 10th IEEE International Conference on Computer Modeling and Simulation, 2008.

[9] R. Buyya, M. M. Murshed, GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing, Journal of Concurrency and Computation: Practice and Experience 14 (13–15).

[10] H. Xia, H. Dail, H. Casanova, A. A. Chien, The microgrid: Using online simulation to predict application performance in diverse grid network environments, in: Challenges of Large Applications in Distributed Environments (CLADE), Honolulu, HI, USA, 2004, p. 52.

[11] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, D. Becker, Scalability and accuracy in a large-scale network emulator, SIGOPS Oper. Syst. Rev. 36 (SI) (2002) 271–284. doi:http://doi.acm.org/10.1145/844128.844154.

[12] Grid explorer (GdX).
URL http://www.lri.fr/~fci/GdX/

[13] The DAS-3 project.
URL http://www.cs.vu.nl/das3/

[14] Planet lab.
URL http://www.planet-lab.org/

[15] A. Takefusa, S. Matsuoka, H. Nakada, K. Aida, U. Nagashima, Overview of a performance evaluation system for global computing scheduling algorithms, in: the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC'99), Redondo Beach, California, USA, 1999.

[16] H. Lamehamedi, Z. Shentu, B. Szymanskia, E. Deelman, Simulation of dynamic data replication strategies in data grids, in: the 17th International Symposium on Parallel and Distributed Processing, Nice, France, 2003, pp. 10pp+.

[17] H. Lamehamedi, B. Szymanski, Z. Shentu, , E. Deelman, Data replication strategies in grid environments, in: Proc. Of the Fifth International Conference on Algorithms and Architectures for Parallel Processing, 2002, pp. 378–383.

[18] The network simulator NS2.
URL http://www.isi.edu/nsnam/ns

[19] J. Prokkola, Opnet - network simulator (2007).
URL http://www.telecomlab.oulu.fi/kurssit/521365A_tietoliikennetekniikan_simuloinnit_ja_tyokalut/Opnet_esittely_07.pdf

[20] A. Varga, The omnet++ discrete event simulation system (2001).
URL http://www.omnetpp.org/download/docs/papers/esm2001-meth48.pdf

[21] P. Vicat-Blanc Primet, O. Glück, C. Otal, F. Echantillac, Emulation d'un nuage réseau de grilles de calcul: eWAN, Tech. Rep. RR 2004-59, LIP ENS, Lyon, France (2004).

[22] The RAMP project: Research Accelerator for Multiple Processors.
URL http://ramp.eecs.berkeley.edu/

[23] iproute2+tc notes: http://snafu.freedom.org/linux2.2/iproute-notes.html.

[24] parXXL: A fine grained development environment on coarse grained architectures.
URL http://parxxl.gforge.inria.fr/

[25] A. V. Gerbessiotis, L. G. Valiant, Direct bulk-synchronous parallel algorithms, Journal of Parallel and Distributed Computing 22 (1994) 251–267.

[26] O. Beaumont, V. Boudet, F. Rastello, Y. Robert, Matrix multiplication on heterogeneous platforms, IEEE Trans. Parallel Distributed Systems 12 (10) (2001) 1033–1051.

[27] A. Lastovetsky, R. Reddy, Data partitioning with a realistic performance model of networks of heterogeneous computers with task size limits, in: Proceedings of the Third International Symposium on Parallel and Distributed Computing / Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (ISPDC/HeteroPar'04), IEEE Computer Society Press, 2004, pp. 133–140.

[28] F. Vernier, Algorithmique itérative pour l'équilibrage de charge dans les réseaux dynamiques, Ph.D. thesis, Univ. de Franche-Comté (2004).

[29] The persistence of vision raytracer.
URL http://www.povray.org/

[30] R. D. Blumofe, C. E. Leiserson, Scheduling multithreaded computations by work stealing, J. ACM 46 (5) (1999) 720–748. doi:http://doi.acm.org/10.1145/324133.324234.

[31] J. Nagle, On packet switches with infinite storage, IETF RFC 970 (1985).
URL http://www.rfc-archive.org/getrfc.php?rfc=970

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 (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
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