Improving the Accuracy and Efficiency of Time-Independent Trace Replay

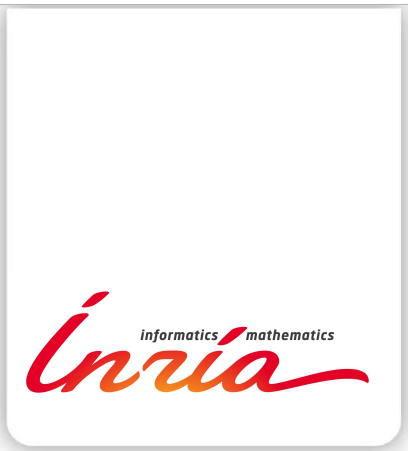
Frédéric Desprez, George Markomanolis, Frédéric Suter

Improving the Accuracy and Efficiency of Time-Independent Trace Replay

Frédéric Desprez, George S. Markomanolis, Frédéric Suter

RESEARCH REPORT
N° 8092
October 2012
Project-Team Avalon

ISSN 0249-6399 ISRN INRIA/RR--8092--FR+ENG



Improving the Accuracy and Efficiency of Time-Independent Trace Replay

Frédéric Desprez, George S. Markomanolis, Frédéric Suter

Project-Team Avalon

Research Report n° 8092 — October 2012 — 17 pages

Abstract: Simulation is a popular approach to obtain objective performance indicators on platforms that are not at one's disposal. It may help the dimensioning of compute clusters in large computing centers. In a previous work, we proposed a framework for the off-line simulation of MPI applications. Its main originality with regard to the literature is to rely on time-independent execution traces. This allows us to completely decouple the acquisition process from the actual replay of the traces in a simulation context. Then we are able to acquire traces for large application instances without being limited to an execution on a single compute cluster. Finally our framework is built on top of a scalable, fast, and validated simulation kernel.

In this paper, we detail the performance issues that we encountered with the first implementation of our trace replay framework. We propose several modifications to address these issues and analyze their impact. Results shows a clear improvement on the accuracy and efficiency with regard to the initial implementation.

Key-words: Performance prediction, MPI, Simulation

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Amélioration de la précision et de l'efficacité du rejeu de traces indépendantes du temps

Résumé : La simulation est une approche populaire pour obtenir des indicateurs de performance objectifs sur des plates-formes qui ne sont pas nécessairement accessibles. Elle peut par exemple aider au dimensionnement d'infrastructures dans de grands centres de calcul. Dans un article précédent, nous avons proposé un environnement pour la simulation hors-ligne d'applications MPI. La principale originalité de cet environnement par rapport à la littérature est de ne reposer que sur des traces indépendantes du temps. Cela nous permet de découpler totalement l'acquisition des traces de leur rejeu simulé effectif. Nous sommes ainsi capables d'obtenir des traces pour de très grandes instances d'applications sans être limités à une exécution au sein d'une seule grappe de machines. Enfin, cet environnement est fondé sur un noyau de simulation extensible, rapide et validé.

Dans cet article nous détaillons les problèmes de performance rencontrés par la première implantation de notre environnement de rejeu de traces. Nous proposons plusieurs modifications pour résoudre ces problèmes et analysons leur impact. Les résultats obtenus montrent une amélioration notable à la fois en termes de précision et d'efficacité par rapport à l'implantation initiale.

Mots-clés : Prédiction de performances, MPI, Simulation

1 Introduction

Understanding the performance and behavior of parallel applications relying on the Message Passing Interface (MPI) on various compute infrastructures is an main concern for many developers. Many tools exist, such as TAU [14], Scalasca [6], or the joint effort on SCORE-P [1], that provide strong analysis methods based on profiling to detect performance bottlenecks.

Simulation is another popular approach for predicting the performance of an application. When a platform is yet to be specified and purchased, simulations can be used to determine a cost-effective hardware configuration appropriate for the expected application workload. Conversely, simulations can also be used to study the performance behavior of an application by varying the hardware characteristics of an hypothetical platform. In a classroom setting, students without access to a parallel platform could execute applications in simulation on a single node as a way to learn the principles of parallel programming and high-performance computing. Simulation of an application on a platform may also be useful even when the platform is available. For instance, the simulation may bypass actual computations performed by the application, and only simulate the corresponding delays of these computations. In this case the simulated application produces erroneous results, but its performance behavior may be preserved. It is then possible to conduct development activities for performance tuning in simulation only on a small-scale platform, thereby saving time when compared to real executions on a large-scale platform. Furthermore, access to large-scale platforms is typically costly (possible access charges to the user, electrical power consumption). The use of simulation can thus not only save time but also money and resources.

Many simulation frameworks have been proposed over the last decade. They fall into two categories: *on-line simulation*, also called simulation via direct execution, and *off-line simulation*, also called post-mortem simulation. In on-line simulation the application is executed but part of the execution takes place within a simulation component. In off-line simulation a trace of a previous execution of the application is "replayed" on a simulated platform. Most of the existing tools in this category rely on timed traces, i.e., each traced event is associated to a time-stamp. This approach creates a tight link between the trace and the set of machines used to produce it. This link is a clear limit to the applicability of the off-line approach. In [5] we proposed a first prototype of a framework that breaks this link by getting rid of the time-stamps. This greater flexibility related to the acquisition of traces comes with some issues of its own. Some were highlighted by the experiments conducted in [5]. First, the acquisition of the traces is sometimes time-consuming and may, in very specific cases, be close to the execution time of the application itself. Then, as our framework relies solely on volumes and not on timestamps, it requires a very accurate measure of hardware counter values, and a minimal impact of the instrumentation. We noted a certain discrepancy of the measured number of instructions in our first implementation that may hinder the quality of the simulation results. Finally, while the accuracy of the simulations is very good for either compute or communication intensive applications, it was not the case for applications that alternate computations and communications.

This paper is organized as follows. In Section 2 we detail the issues raised by the first prototype of our off-line simulation framework based on Time-Independent Traces. In Section 3 we present the different modifications of the framework aiming at addressing these issues. The impact of the proposed modifications are given in Section 4. We review related work in Section 5. Finally, we draw some conclusions and detail future work directions in Section 6.

2 Identifying the Issues

In this section, we analyze the performance of our Time-Independent Trace replay framework. We detail the different issues that raised in the evaluation section of [5]. We reproduce experiments conducted on the same compute cluster, called *bordereau*, that comprises 93 2.6 GHz Dual-Proc, Dual-Core AMD Opteron

2218 nodes interconnected through a single 10 Gigabit switch. Each core has a L2 cache of 1 MB. As this cluster is now aging and is prone to failures and suspect behaviors, we complement this study with results obtained on a more recent cluster, called *graphene*, that comprises 144 2.53GHz Quad-Core Intel Xeon x3440 nodes. The L2 cache owned by each core is two times larger than on the *bordereau* cluster. The nodes are scattered across four cabinets, and interconnected by a hierarchy of 10 Gigabit switches.

2.1 Instrumentation Time Overhead

In our first implementation of a Time-Independent Trace replay framework, we relied on the TAU profiling tool [14] and its Program Database toolkit (PDT) [9] to trace the execution of an MPI application. We also took advantage of the *selective instrumentation* feature offered by TAU to focus on specific parts of the studied application. Using such tools was very convenient as the instrumentation is done automatically. However, the studied application is instrumented at a very fine grain. This may lead to a non-negligible, overhead on the execution time as shown in [5]. Basically, this overhead is the difference between the respective execution times of an application with and without instrumentation. As identified in [11], this overhead has different sources, e.g., intrusion of the measurement mechanisms, building the complete call path of the application and flushing the trace on disk.

In [5], we measured an overhead ranging from 4.1% to 39.5%, depending on the problem instance, for the execution of a LU factorization. The largest value was observed for small problem sizes executed on a large number of compute resources. While this overhead remains acceptable as traces are acquired once to be replayed multiple times, we aim at reducing it as much as possible. On the *graphene* cluster, this overhead ranges from 9.8% to 25.5% in similar conditions. For 128 processes, the overhead increases to 37.9%.

2.2 Impact of Instrumentation on Hardware Counters Values

A second issue is related to the impact of the instrumentation on the values of the hardware performance counters used to produce a Time-Independent Trace. In our work, we use the total number of instructions executed by the studied application. Adding instrumentation probes may increase this value and then impact the accuracy of a simulation based on the produced trace. To measure the impact of a fine grain instrumentation such as that done by TAU, we perform the following simple experiment. We execute two versions of the same application, namely the LU factorization, on the same compute cluster. The first version is fully instrumented using TAU. In the second version, we just insert calls to get the value of the hardware performance counter that measures the number of executed instructions at the beginning and end of the studied section of the code. Figure 1 shows the distribution across processes of the relative difference in terms of measured numbers of executed instructions between fine and coarse grain instrumented versions of various instances of the LU factorization. The comparison between the two executions is done on a per-process basis. Moreover, we ran ten runs of each version and display the average values. Finally the average total number of instructions per process measured with the coarse grain instrumentation method is 1.70e11 for the smallest instance (B-8) while it is 8.87e10 for the largest instance (C-64).These results were obtained on the *bordereau* cluster.

With the exception of instance B-64, we see that the fine grain instrumentation leads to an increase of the measured number of instructions of around 10-13%. As our Time-Independent Traces are directly using these numbers of instructions, such an increase will have a negative impact on the accuracy of the simulation. In other words, it will likely simulate something closer to the instrumented version than the original application. The discrepancy becomes even worse when communications start to prevail. This is the case with the B-64 instance in which the balance between computation and computation changes due to data distribution.

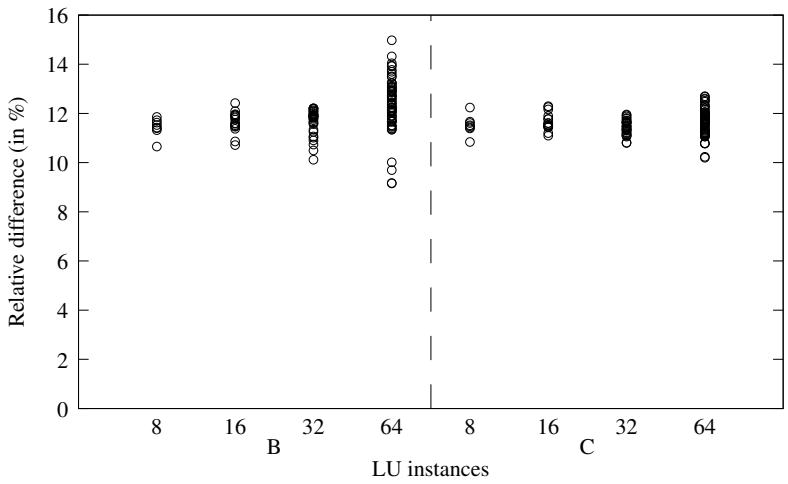


Figure 1: Distribution across processes of the relative difference (in %) of the measured numbers of executed instructions between fine and coarse grain instrumented versions of various instances of the LU factorization on the *bordereau* cluster.

Figure 2 presents similar results but obtained on the more recent *graphene* cluster. For instances that involves between 8 and 64 processes, we note a similar discrepancy on this cluster as on the *bordereau* cluster. It ranges from 11 to 16%. However, we observe a clear increase of the discrepancy as the number of processes grows for the Class B instances. For 128 processes, it raises up to 23%. It can easily be explained as, for such an instance, the rather small input data is highly distributed. Each process thus only has a really small amount of data to compute on and is more impacted by the instrumentation.

2.3 Calibration and Cache Usage

An essential step to make accurate performance predictions through trace replay is the calibration of the simulation framework. In our framework, it consists in determining the number of instructions a CPU can compute in one second and the latency and bandwidth of communication links. Such a calibration strongly depends on both application and execution environment.

In our first implementation, we chose to do the calibration based on the execution of a small instance of the studied application. For instance, we consider the A-4 instance to calibrate our tool for the simulation of larger instances of the LU factorization. The problem there is that the A-4 instance is small enough to have all data stored in the L2 cache. Then it does not allow us to capture the worse performance achieved by instances for which the data distribution leads to data movements between L2 cache and main memory.

The instruction rate being one the most important parameter to achieve accurate simulated executions, we have to define a more complex calibration step to better take cache usage in account.

2.4 Accuracy of the Simulated Execution

In the evaluation of the performance of our off-line simulation framework we conducted in [5], we compared the execution time of various instances of the LU factorization to the simulated time obtained by replaying Time-Independent Traces (see Fig. 8 in [5]). For such an application that is composed of millions of interleaved computation and communication actions, the accuracy of the simulated replay was not as good as expected.

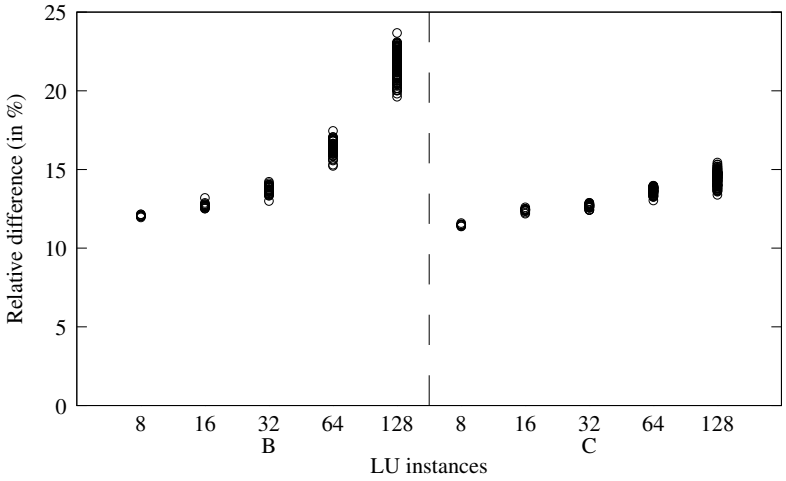


Figure 2: Distribution across processes of the relative difference (in %) of the measured numbers of executed instructions between fine and coarse grain instrumented versions of various instances of the LU factorization on the *graphene* cluster.

Figure 3 shows the evolution of the relative error between the time to compute a LU factorization and its simulated counterpart when the number of involved processes vary. We see that the inaccuracy of the simulated version is not stable, but increases rather linearly with the number of processes. Moreover, our tool underestimates the execution time for small number of processes, i.e., when each process owns a larger share of data, and then overestimates the execution as the number of processes grows.

This inaccuracy may have different origins related either to computations or communications. Such an inaccuracy may for instance come from the discrepancy of the measured number of instructions that we just mentioned. Indeed it directly impacts the calibration of the replay tool that determines the rate at which each machine can process instructions. Among other potential sources, we assume that every part of the application can be processed at the same rate which may not be the case for some applications, due to cache affinity for instance. However, the general trend shown by Figure 3 indicates that the main source of inaccuracy comes from a bad estimation of communication times. The most probable source might be that the LU factorization, as implemented in the NAS Parallel Benchmark suite, implies a lot of small messages (less than 64 KiB). Then most of the point-to-point communications use the *eager mode* of MPI in which the sender copies the data directly into the memory of the receiver without having to wait for it to post the corresponding receive. This particular communication protocol was implemented in a too simple way, based on asynchronous communications for small messages, in the first version of our framework. As the number of small messages increases with the number of involved processes, small inaccuracies tend to accumulate and lead to a large overall relative error.

3 Addressing the Issues

In the previous section we have highlighted four issues that all affect the performance and the quality of our off-line simulation framework. We detail hereafter four approaches to address these issues. The first lever we use is external to our framework, i.e., leveraging compiler optimizations. The other three imply modifications of the first implementation presented in [5], i.e., reducing the intrusiveness of the instrumentation method, changing the simulation back-end to better handle communications, and improving the calibration procedure.

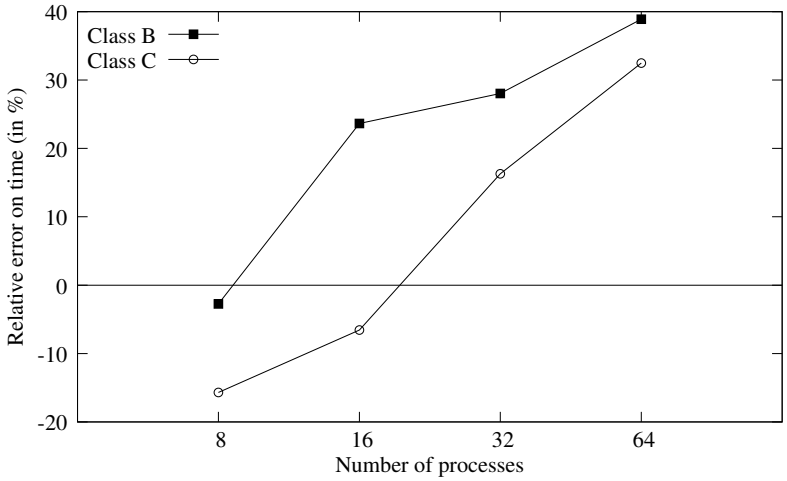


Figure 3: Evolution with regard to the number of processes of the relative error between execution and simulated times for the execution of the LU factorization.

3.1 Compiler Optimizations

The first modification we made to our trace acquisition procedure is to activate compiler optimizations, typically by using the `-O3` flag. Turning on an optimization flag makes the compiler attempt to improve the performance and/or code size at the expense of compilation time. Among the optimizations that may help to reduce the discrepancy in the measured number of instructions are the loop unrolling, vectorization, and function inlining.

3.2 Less Intrusive Instrumentation Method

As stated in Section 2.1, the default instrumentation provided by TAU leads to some execution time overhead. The main source of this overhead comes from the building of the complete call path of the instrumented application. While this information may be useful to users aiming at identifying performance bottlenecks in their application, it is useless to our specific usage of the produced traces. Indeed, the only information required by our Time-Independent Trace replay framework are: (i) the volume of computation, in number of instructions executed between two MPI calls; (ii) the name and parameters of each MPI call; and (ii) the volume of data transferred by each communication operation. This is illustrated by the trace snippet below.

```
p0 compute 956140
p0 send p1 1240
p0 compute 2110
p0 send p2 1240
p0 compute 3821
```

The selective instrumentation feature provided by TAU offers us a way to obtain all this required information, and only it. Then we modify our instrumentation method as follows. First we create a file, named `exclude.pdt`, in which we indicate to TAU the list of source files that have to be excluded from the detailed instrumentation. Actually, we exclude all the source files using the special character `*`.

```
BEGIN_FILE_EXCLUDE_LIST
*
END_FILE_EXCLUDE_LIST
```

Then, we add the following option to the command line to take this exclusion file into account.

```
-optTauSelectFile=/path/exclude.pdt
```

Thanks to this straightforward modification, the instrumentation becomes *minimal* with regard to our specific needs. Indeed the performance hardware counter that measures the number of executed instructions will be triggered when entering and exiting MPI functions. All the information related to MPI calls, i.e., id of the process that calls this function and the function name and calling parameters, are traced exactly as in the previous implementation.

We expect this simple modification to have an impact both on the execution time overhead and the discrepancy of measured numbers of instructions between non-instrumented and instrumented versions of an application.

3.3 Simulation Back-end

Our simulation framework is is tightly connected to the SIMGrid project [3] that provides core functionalities for the simulation of distributed applications in heterogeneous distributed environments. SIMGrid relies on a scalable and extensible simulation engine and offers several user APIs. Our first implementation of a trace replay mechanism was based on the MSG API that was designed to implement simulators using Concurrent Sequential Processes (CSP). This choice forced us to mimic the behavior of MPI primitives, e.g., collectives operations or protocols depending on the message size, with some crude simplifications. More importantly it decouples this effort on the off-line simulation of MPI applications to that on the on-line simulation of MPI applications also taking place within the SIMGrid project.

In [4], we proposed a new simulator, SMPI, for the on-line approach. It is, as the present off-line approach, built on top of the simulation kernel of the SIMGrid toolkit and thus also benefits of its fast, scalable, and validated network models. SMPI extends the existing models with an original piece-wise linear model to take into account the specifics of the cluster interconnect. A salient property of SMPI is its capacity to simulate MPI applications on a single node.

We have reimplemented the trace replay mechanism directly within SMPI, instead of on top of the MSG API. This rewriting leads to many beneficial changes. First, it simplifies the way *actions*, i.e., events stored in the trace are passed to the simulation kernel. To illustrate this, we show the evolution of the implementation of the action that corresponds to an `MPI_Send` call. The format of this action is `<id> send <dst_id> <volume>`.

```
static void action_send(const char *const *action){
 char to[250];
 double size=parse_double(action[3]);

 sprintf(to, "%s_%s", MSG_process_get_name(MSG_process_self()), action[2]);

 if (size<65536)
  action_Isend(action);
 else
  MSG_task_send(MSG_task_create(NULL, 0, size, NULL), to);
}
```

The underlying principle there is to create a *mailbox* named after the ids of the sender and receiver processes, create a task that corresponds to the communication of a message, and then send this task into that mailbox. A matching action on the receiver side, will read the contents of the mailbox and execute the task, which actually starts the simulated communication. This implementation does not reflect some the MPI specifics, especially for small messages. Indeed, when the message is smaller than 64KB, the *eager mode* is activated. This means that the sender does not have to wait for the receiver to allocate memory and directly copies the data remotely. We tried to model that by using an asynchronous send for such small messages. However, it is not what is actually implemented by most MPI runtimes.

Through the analyze of execution traces, we found that an `MPI_send` for small messages acts in a sort of "detached" way. From the application point of view, the send corresponds to the time of a copy of the data in the memory. Moreover, if the receive is issued after the send, the data is already stored in memory. There again, the application only sees the duration of a memory copy. Modeling this particular behavior has been done in SMPI. Doing the same in our initial trace replay tool would have been redundant, error prone, and above all difficult with the MSG API. The new implementation allows us to benefit of all the complex optimizations offered by SMPI in a seamless way. Indeed the send action is now:

```
static void action_send(const char *const *action){
 int to = atoi(action[2]);
 double size=parse_double(action[3]);

 smpi_mpi_send(NULL, size, MPI_BYTE, to, 0, MPI_COMM_WORLD);
}
```

The second main change implied by this rewriting is the user view of the replay framework. In the former version everything was exposed to the user. Everything is now embedded and replaying a Time-Independent Trace with SimGrid simply amounts to run the following program.

```
int main(int argc, char *argv[]){
 smpi_replay_init(&argc, &argv);
 smpi_action_trace_run(NULL);
 smpi_replay_finalize();
 return 0;
}
```

This code, that initializes some data structures, loads a trace, and destroys the data structures, is considered as a regular SMPI program. Then it is compiled with the `smpicc` wrapper and launched thanks to the `smpirun` command (see [4] for details), as follow

```
smpirun -np 8 -hostfile hostfile -platform platform.xml \
        ./smpi_replay trace_description
```

where `np` and `hostfile` are classical parameters of `mpirun`. In addition, SMPI requires a description of the simulated platform, given in the `platform.xml` file. Our replay tool needs a single parameter, a file that list the names of the trace files to associate to each process. If this file contains a single entry, all the processes will look for the actions they have to perform into the same trace.

The last important change is about the format of the `recv` action. To keep our implementation as simple as possible, we had to add the message size to the parameters of this action. As we were already able to track this information in the traces produced by TAU, it required only a small modification of the existing extraction script.

3.4 Cache-Aware Calibration Step

As mentioned in Section 2.3, the first implementation of our framework relied on small instances, typically Class A on 4 processes for the NAS Parallel Benchmark suite, to calibrate the instruction rate of the simulated platform. The rationale was to limit the number of resources and the time needed to do the calibration as much as possible. Experiments shown that while using only as few resources as four cores did not raise any issue, limiting the calibration to a small problem size did hide some important phenomena.

In the particular case of the LU factorization, as soon as the share of the matrix owned by each process exceeds the capacity of the L2 cache, the performance drops, with a direct impact on the instruction rate. To circumvent this issue, we propose to complete our calibration procedure, when it is needed, with runs on larger problem sizes but on the same number of resources. In practical terms, we run the B-4 and C-4 instances in addition to the A-4 instance, and thus determine three different instruction rates, one per studied class. Then, depending on whether the current instance handles data that fit in the L2 cache or not, we use the rate coming from the A-4 calibration or the one that corresponds to the instance class.

Note that the use of this cache-aware calibration is mandatory only for the *bordereau* cluster whose L2 cache is small (1MB per core only). On the *graphene* cluster that has a L2 cache of 2MB per core, all the instances do fit in cache. Calibrating the simulator with a run of the A-4 instance is then enough.

4 Impact of the Modifications

Here we evaluate the impact of the modifications detailed in the previous section on each of the issues identified in Section 2.

4.1 Instrumentation Time Overhead

Two of the proposed modifications have an impact on the overhead implied by the instrumentation of the application. Applying the `-O3` optimization flag transforms the source code and reduces the execution time of both non-instrumented and instrumented versions of the application. Reducing the instrumentation to its minimum will reduce its intrusiveness and thus hinder its impact.

Table 1: Evolution on the *bordereau* cluster of the execution time and overhead of original and instrumented versions of instances of the LU factorization between the former implementation in [5] and the modified implementation (`-O3` and minimal instrumentation).

	Execution times as in [5]		Execution times after modifications	
	Orig.	Instr.	Orig.	Instr.
B-8	93.05s	98.64s (+6%)	76.55s	86.27s (+13.7%)
B-16	41.49s	47.09s (+13.5%)	35.43s	39.28s (+10.9%)
B-32	23.75s	29.61s (+24.7%)	20.84s	25.43s (+22.02%)
B-64	14.86s	20.73s (+39.5%)	13.71s	17.82s (+30%)
C-8	393s	409.27s (+4.1 %)	356.49s	363.42s (+1.9%)
C-16	193s	210.63s (+9.1 %)	166.6s	186.8s (+12.1%)
C-32	86.95s	104.50s (+20.2%)	73.90s	83.38s (+12.82%)
C-64	46.76s	57.77s (+23.5%)	41.03s	47.76s (+16.4%)

Table 1 shows the evolution of the execution times of the original application and its instrumented version between the first implementation presented in [5] and the current one that implements both modifications. Note that the performance achieved by the current implementation may also be impacted by an update of TAU from version 2.18.3 to version 2.21.1. First, we can note a clear reduction of the execution times of both non instrumented and instrumented versions, coming mainly from the `-O3` optimization flag. While simple, this optimization is important as it shortens the overall acquisition time of traces. This gain will become even more interesting for larger problem sizes. If we focus on the evolution of the instrumentation overhead, we see that the impact of the proposed modifications is not stable. This overhead actually increases for two instances (B-8 and C-16) and no clear evolution trend can be found. However, the gain is the most significant for instances that were the most impacted by the instrumentation, justifying the introduction of this modifications. Over the complete set of instances, the instrumentation overhead now ranges from 1.9% to 30% instead of the [4.1%-39.5%] range mentioned in Section 2.

As before, we complete the study on the impact of the proposed modification with results obtained on a more recent compute cluster. We ran experiments with the same initial settings on the *graphene* cluster, whose results are shown by Table 2.

Table 2: Evolution on the *graphene* cluster of the execution time and overhead of original and instrumented versions of instances of the LU factorization between the former implementation in [5] and the current modified implementation (`-O3` and minimal instrumentation).

	Execution times as in [5]		Execution times after modifications	
	Orig.	Instr.	Orig.	Instr.
B-8	50.96s	58.58s (+14.95%)	40.80s	41.09s (+0.71%)
B-16	27.77s	32.58s (+17.32%)	22.66s	23.26s (+2.61%)
B-32	16.53s	20.11s (+21.26%)	14.04s	14.95s (+6.48%)
B-64	10.24s	12.85s (+25.49%)	8.82s	10.11s (+14.62%)
B-128	7.71s	10.631s (+37.88%)	6.89s	8.76s (+27.14%)
C-8	203.3s	223.13s (+9.84 %)	153.16s	153.40s (+0.16%)
C-16	102.98s	119.62s (+16.16 %)	82.13s	83.14s (+1.23%)
C-32	56.63s	65.80s (+16.2%)	45.91s	47.07s (+2.52%)
C-64	31.69s	37.68s (+18.9%)	26.22s	27.94s (+6.55%)
C-128	23.14	28.32s (+22.39%)	19.45s	22.07 (+13.47%)

As on the *bordereau* cluster, the use of the `-O3` flag has a direct positive impact on the execution time of the application. But the reduction of the instrumentation overhead is more important there. With the first implementation of our framework, the overhead goes from 9.8% to 37.9%, which is consistent with the experiments on the *bordereau* cluster. The proposed modifications make the overhead range shrink to [0.7% - 27.1%]. More interestingly, we can see a stable growth of this overhead as the number of processes increases. As on the *bordereau* cluster, the improvement is smaller for large number of processes, but remains significant.

4.2 Impact of Instrumentation on Hardware Counters Values

To measure the impact of the proposed modifications, i.e., applying the `-O3` optimization flag and minimizing the intrusiveness of the instrumentation, on the discrepancy of the measured number of instructions, we made the same experiments as in Section 2.2. Figure 4 shows results obtained on the *bordereau* cluster.

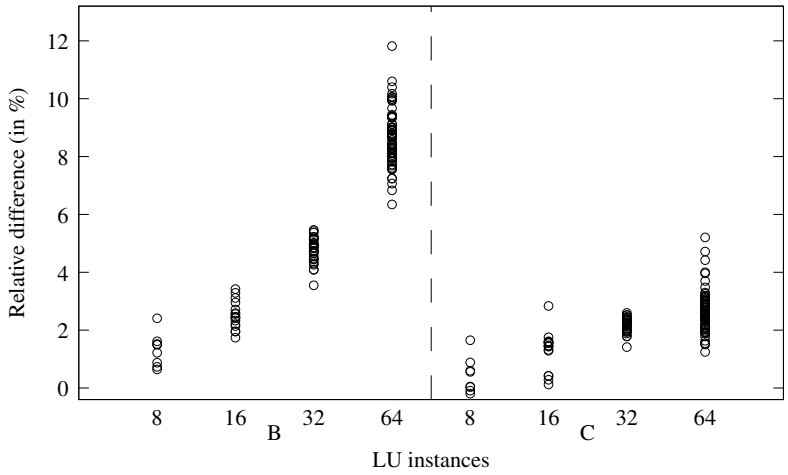


Figure 4: Distribution across processes of the relative difference of the measured numbers of executed instructions between minimal and coarse instrumentations of optimized (`-O3`) LU instances. Results obtained on the *bordereau* cluster.

For most instances, the discrepancy is greatly reduced. Except for the B-64 instance, all are now under 6% while they were around 10-13% before. Moreover, we now see the same evolution as in Figure 2 even on the *bordereau* cluster. This means that the discrepancy now becomes really significant only when the amount of data handled by each process is small. Nevertheless, for the B-64 instance, there is still a significant reduction from 16% to 12% for the worst difference of measurement between both versions.

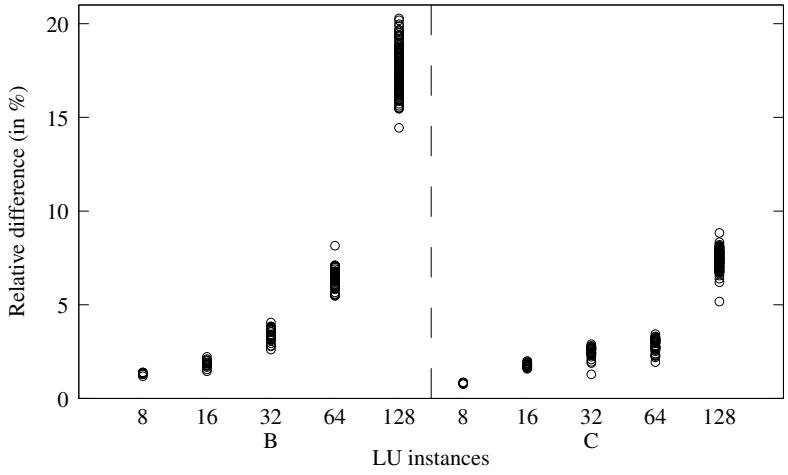


Figure 5: Distribution across processes of the relative difference of the measured numbers of executed instructions between minimal and coarse instrumentations of optimized (`-O3`) LU instances. Results obtained on the *graphene* cluster.

These results are confirmed on the more recent *graphene* cluster, as shown by Figure 5. The discrepancy is reduced by at least 5% for all instances, most of them being even close to zero. The modifications accentuate the discrepancy increase caused by the evolution of the number of processes. But as for the overhead in terms of execution, instances such as B-128 are not representative of typical executions of

parallel applications. Indeed, the benefit of distributing the computation comes to its limits for such instances, with too few data to compute on for each process. Consequently results for these instances are less significant than for instances with a better balance of the computation over communication ratio.

These experiments clearly show the benefits of the proposed modifications on the quality of the instrumentation. We expect it to contribute to the improvement of the accuracy of the simulated execution, studied in the next Section, once combined to a better calibration and the change of simulation back-end.

4.3 Accuracy of the Simulated Execution

In this last set of experiments we combine all the proposed modifications. We rely on the compiler optimization flag, the minimal instrumentation, and the cache-aware calibration method to improve the quality of the simulation of the compute part of the application. For the communication part, we benefit of the quality of the underlying SMPI layer that provides good estimations for the point-to-point communication of small messages and a tuned piece-wise linear network model.

Figure 6 shows the impacts of the whole set of modifications on the accuracy of the simulated executions of the LU factorization. These results, obtained on the *bordereau* cluster, have to be compared to those presented by Figure 3.

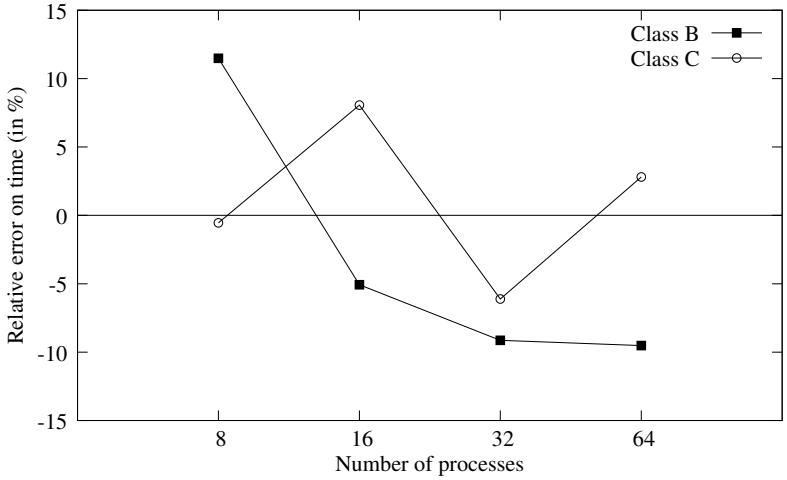


Figure 6: Evolution w.r.t. the number of processes of the relative error between execution and simulated times for the execution of the LU factorization with the new replay framework. Results obtained on the *bordereau* cluster.

The first outcome is a drastic reduction of the inaccuracy. With our initial implementation, the relative error linearly increased from -2.7% to 38.9% for Class B and from -15.8% to 32.5% for Class C. Now, the error is comprised between -9.5% and 11.5% for Class B and between -6.1% and 8.1% for Class C. Moreover, the linear increase of the error along with the number of processes, that indicated a bad simulation of the communication part, disappeared. We even note an opposite trend for Class B. This can be explained by the fact that SMPI does not model the time to copy data in memory in the `MPI_Send` function yet. This leads to a slight underestimation of the simulation time that increases as more small messages are exchanged for large number of processes. This phenomenon is well identified and will soon be addressed.

Figure 7 shows the relative error achieved under the same experimental conditions on the *graphene* cluster. As on the *bordereau* cluster, the error is in a narrow interval ranging from –11.4% and -2%.

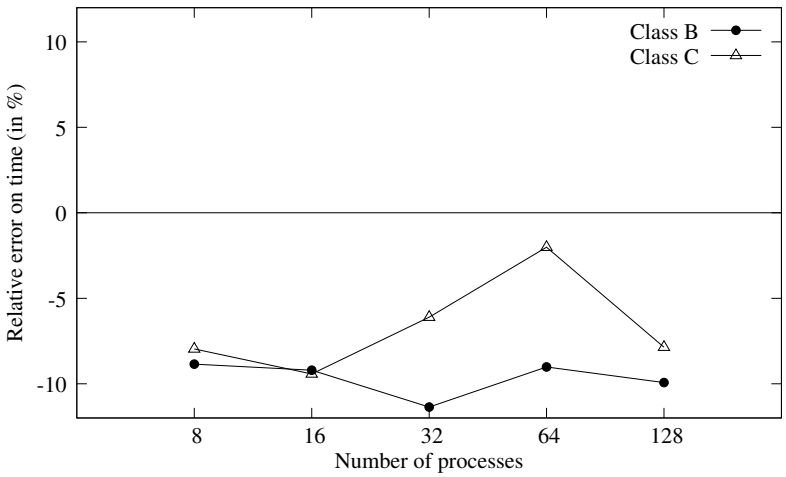


Figure 7: Evolution w.r.t. the number of processes of the relative error between execution and simulated times for the execution of the LU factorization with the new replay framework. Results obtained on the *graphene* cluster.

Results for Class B are even more stable. Again the underestimation of the simulated time should be compensated by taking memory copy into account.

While narrowing the inaccuracy range is a good result in itself, the more important result shown by Figures 6 and 7 is the greater stability of this relative error. Indeed, the strong linear increase highlighted by Figure 3 prevented us to use our framework for its main objective, i.e., predicting the execution time of any instance of a given application within a certain confidence interval.

The results presented in this section are not perfect yet, as some fluctuations or linear trends are still there, but there are a great improvement towards an accurate and trustworthy performance prediction tool. Some issues are still to be addressed, but most of them have been identified. Finally the factorization of the code, and thus of the efforts with those made on the SMPI project will help to improve the quality of our Time-Independent Trace replay tool.

5 Related Work

The off-line simulation of MPI application has been used extensively, as shown by the number of trace-based simulators described in the literature since 2009 [8, 15, 12, 16, 7].

The typical approach is to decompose the trace in time intervals delimited by the MPI communication operations. The application is thus seen as a succession of computation and communications steps. Most of the existing tools then simply replace the computations by the measured delays. The performance differential between the platform used to obtain the trace and the target platform can be taken into account, but usually using a simple scaling [15, 12, 7]. Network communications are simulated based on the communication events recorded in the trace and on a simulation model of the network. As for computation, most of the tools adopt simplistic network models. The most common simplifications are to ignore network contention because it is known to be costly to simulate [17], and use monolithic performance models of collective communications rather than simulating them as sets of point-to-point communications [15, 2]. Exception is the MPI-NetSim on-line simulator [13], which relies on a slow packet-level discrete event network simulator.

6 Conclusion and Future Work

Simulation is a popular approach to obtain objective performance indicators platforms that are not at one's disposal. It may help the dimensioning of compute clusters in large computing centers. In [5], we proposed a new approach for the off-line simulation of MPI applications. Instead of relying on logs of execution that associate an event to a time-stamp, we use Time-Independent Traces as an input of our simulator. These traces contain only information about volumes of computation and communications. This allows us to decouple the acquisition process from the replay of the trace. Heterogeneous and distributed platforms can then be used to get traces without impacting the quality of the simulation, which is not possible with any other tool. Our simulator is built on top of the SIMGrid toolkit and thus benefits of its fast, scalable, and validated simulation kernel. We also rely on a well established tracing tool, TAU, in the acquisition process.

In this paper, we identified several issues, i.e., an instrumentation time overhead, a discrepancy of the measured number of instructions, a calibration method that ignores some cache related effects, and an improvable simulation back-end, related to the first implementation of this tool. These issues lead to inaccurate and unstable simulated times and prevent a use for performance prediction within a confidence interval. We proposed several modifications of our framework to address these different issues and evaluated their respective impact. While not perfect, the resulting implementation is far more accurate and stable than the previous one, and becomes a good candidate to predict the performance of MPI applications on platforms that are not available.

This Time-Independent Trace replay framework is freely distributed as part the SIMGrid project under the LGPL license. It can be downloaded from `http://simgrid.gforge.inria.fr/download.php`. The implementation described in this article will be released in the version 3.8 of SIMGrid. Moreover, the process to acquire a Time-Independent Trace has been fully described in [10].

As future work, we plan to implement the missing feature to model the time taken in sends and receives to copy data in memory in the eager mode of MPI. We also aim at improving our calibration method to automatically take cache usage into account and better estimate the instruction rate used by the simulator. Finally, we would like to assess the performance of the proposed approach on an MPI application used in production.

Acknowledgments

This work is partially supported by the ANR project SONGS (11 ANR INFRA 13) and the CNRS PICS N° 5473. 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`). The authors would also like to thanks all the members of the SIMGrid project working on the SMPI effort, and especially A. Degomme and L. Schnorr for their great help.

References

[1] D. an Mey, S. Biersdorff, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, C. Rössel, P. Saviankou, D. Schmidl, S. S. Shende, M. Wagner, B. Wesarg, and F. Wolf. Score-P –A Unified Performance Measurement System for Petascale Applications. In *Proc. of Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V.*, pages 1–12, Schwetzingen, Germany, June 2010. Springer.

[2] R. Badia, J. Labarta, J. Giménez, and F. Escalé. Dimemas: Predicting MPI applications behavior in Grid environments. In *Proc. of the Workshop on Grid Applications and Programming Tools*, 2003.

[3] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *Proc. of the 10th IEEE International Conference on Computer Modeling and Simulation*, Cambridge, UK, Mar. 2008.

[4] P.-N. Clauss, M. Stillwell, S. Genaud, F. Suter, H. Casanova, and M. Quinson. Single Node On-Line Simulation of MPI Applications with SMPI. In *Proc. of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Anchorage, AK, May 2011.

[5] F. Desprez, G. S. Markomanolis, M. Quinson, and F. Suter. Assessing the Performance of MPI Applications Through Time-Independent Trace Replay. In *Proc. of the 2nd International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*, pages 467–476, Taipei, Taiwan, Sept. 2011.

[6] M. Geimer, F. Wolf, B. Wylie, and B. Mohr. A Scalable Tool Architecture for Diagnosing Wait States in Massively Parallel Applications. *Parallel Computing*, 35(7):375–388, 2009.

[7] M.-A. Hermanns, M. Geimer, F. Wolf, and B. Wylie. Verifying Causality between Distant Performance Phenomena in Large-Scale MPI Applications. In *Proc. of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 78–84, Weimar, Germany, Feb. 2009.

[8] T. Hoefler, C. Siebert, and A. Lumsdaine. LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model. In *Proc. of the ACM Workshop on Large-Scale System and Application Performance*, pages 597–604, Chicago, IL, June 2010.

[9] K. Lindlan, J. Cuny, A. Malony, S. Shende, B. Mohr, R. Rivenburgh, and C. Rasmussen. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. In *Proc. of SuperComputing 2000*, Dallas, TX, Nov. 2000.

[10] G. S. Markomanolis and F. Suter. Time-Independent Trace Acquisition Framework – A Grid'5000 How-to. Technical Report RT-0407, Institut National de Recherche en Informatique et en Automatique (INRIA), Apr. 2011.

[11] J. Mussler, D. Lorenz, and F. Wolf. Reducing the overhead of direct application instrumentation using prior static analysis. In *Proc. of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 65–76, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] A. Núñez, J. Fernández, J.-D. Garcia, F. Garcia, and J. Carretero. New Techniques for Simulating High Performance MPI Applications on Large Storage Networks. *Journal of Supercomputing*, 51(1):40–57, 2010.

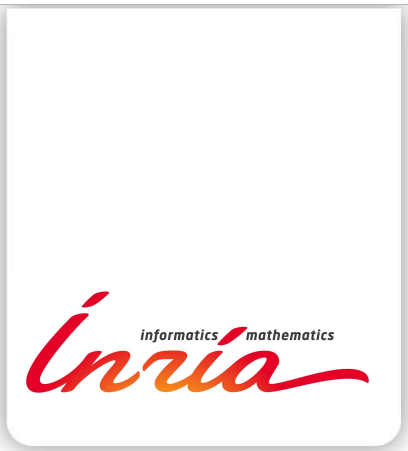
[13] B. Penoff, A. Wagner, M. Tuexen, and I. Ruengeler. MPI-NeTSim: A network simulation module for MPI. In *Proc. of the 15th IEEE International Conference on Parallel and Distributed Systems*, Shenzen, China, Dec. 2009.

[14] S. Shende and A. Malony. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[15] M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely. PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications. In *Proc. of the 15th International EuroPar Conference*, volume 5704 of *LNCS*, pages 135–148, Delft, Aug. 2009.

[16] J. Zhai, W. Chen, and W. Zheng. PHANTOM: Predicting Performance of Parallel Applications on Large-Scale Parallel Machines Using a Single Node. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 305–314, Jan. 2010.

[17] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. Kalé. Simulation-Based Performance Prediction for Large Parallel Machines. *Int. Journal of Parallel Programming*, 33(2-3):183–207, 2005.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399