# Moca: An efficient Memory trace collection system

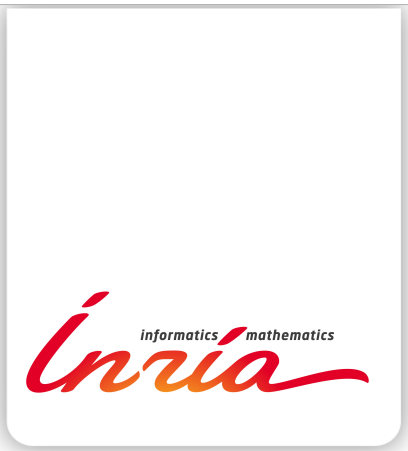
David Beniamine, Guillaume Huard


# Moca: An efficient Memory trace collection system

**David Beniamine, Guillaume Huard**

**RESEARCH REPORT**
**N° 8931**
July 2016
Project-Teams Polaris

ISSN 0249-6399 ISRN INRIA/RR--8931--FR+ENG




# Moca: An efficient Memory trace collection system

David Beniamine, Guillaume Huard

Project-Teams Polaris

Research Report n° 8931 — July 2016 — 16 pages

**Abstract:**
In modern *High Performance Computing* architectures, the memory subsystem is a common performance bottleneck. When optimizing an application, the developer has to study its memory access patterns and adapt accordingly the algorithms and data structures it uses. The objective is twofold: on one hand, it is necessary to avoid missuses of the memory hierarchy such as false sharing of cache lines or contention in a NUMA interconnect. On the other hand, it is essential to take advantage of the various cache levels and the memory hardware prefetcher.
Still, most profiling tools focus on CPU metrics. The few of them able to provide an overview of the memory patterns involved by the execution rely on hardware instrumentation mechanisms and have two drawbacks. The first one is that they are based on sampling which precision is limited by hardware capabilities. The second one is that they trace a subset of all the memory accesses, usually the most frequent, without information about the other ones.
In this study we present *Moca* an efficient tool for the collection of *complete* spatiotemporal memory traces. It is based on a Linux kernel module and provides a coarse grained trace of a superset of all the memory accesses performed by an application over its addressing space during the time of its execution. The overhead of *Moca* is reasonable when taking into account the fact that it is able to collect *complete* traces which are also more precise than the ones collected by comparable tools.

**Key-words:** Moca, Performance evaluation, Memory, Memory traces, NUMA, Cache

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

# Moca: Un système efficient de collecte de traces mémoire

**Résumé :** Dans les architectures de calcul hautes performances, le système de mémoire est une cause fréquente de baisse de performances. Afin d'optimiser une application le.a développeur.euse doit étudier le schéma d'accès mémoire de son application et adapter ses algorithmes et structures de données en conséquence. L'objectif est double : tout d'abord il est nécessaire d'éviter les mauvaise utilisations de la hiérarchie mémoire telles que le faux partage de ligne de cache ou la contention dans les interconnexion *NUMA*. Ensuite il est primordial de tirer le meilleur parti des différents niveaux de cache et du pré-chargement mémoire matériel.

Cependant, la plupart des outils d'analyse de performances se concentrent sur des métriques provenant du processeur. Les rare outils capables de proposer une vue générale des schémas d'accès mémoire se basent sur des mécanismes d'instrumentation matériels et soulèvent deux problèmes. Premièrement ils sont basés sur un échantillonnage dont la précision est limitée par les capacités du matériel. Ensuite ils ne tracent qu'une sous partie des accès mémoire, en général les plus fréquents, sans informations sur les autres accès.

Dans cette étude, nous présentons *Moca* un outil efficace de collecte de traces mémoire spatiotemporelles *complètes*. Cet outil est basé sur un module noyau Linux et génère une trace à gros grain contenant un surensemble des accès mémoire effectués par un application au cours du temps et de l'espace d'adressage de l'exécution. Le surcout de *Moca* est raisonnable si on prend en compte le fait que la trace produite est *complète* et donc plus précise que celles produites par des outils comparable.

**Mots-clés :** Moca, Evaluation de performances, Mémoire, Trace mémoire, NUMA, Cache

# 1 Introduction

In *High Performance Computing* the memory subsystem is often a performance bottleneck. Using efficiently this memory subsystem can be extremely complex, indeed the developer has to take into account both the cache hierarchy and hardware mechanisms such as the memory prefetcher. It gets even more complex with *Non Uniform Memory Access* machines in which data location in physical memory impacts the accesses latency and contention issues arise [1].

Most efforts about memory performance optimizations at software level consists on tools such as NUMA Balancing [2]. These tools automatically moves pages of data in the physical memory, trying to keep them as close as possible from the threads that use them. Such optimizations can improve significantly the memory performances as they reduce the number of remote accesses. Still this cannot fix the application when its algorithms and data structures do not make a proper use of memory resources. For instance, if all the threads of an application access to the same memory page, whatever the page mapping is, this will result in remote accesses from all NUMA nodes but one. The only way to fix this kind of issues is to rewrite a part of the application code in order to take into account these unbalanced memory accesses. This is why tools that can help the developer understand the memory access patterns performed by his application are of utter importance for performance optimizations.

Analysis tools such as Vtune [3] and HPC-Toolkit [4] have been developed to help the programmer understand and correct performance issues in a multithreaded application. However, these tools focus on the CPU and can only trace an indirect and incomplete view of events related to the memory: the location of accesses, their time or both are usually lost in the process. Thus, they are not able to provide the developer with a clear view of memory accesses patterns occurring during the execution. In such situation, fixing memory related issues is a matter of trial and error and the eventual code is often suboptimal.

To solve memory related issues, the developer needs a detailed trace of memory accesses at a sufficiently fine *granularity*. An ideal tool should provide enough data to build a map of the memory accesses locations over the time. Moreover, to ensure that a lack of precision does not compromise the analysis, such a trace should be *complete*. We say that a trace is *complete* at a certain granularity if and only if the events it contains form a superset of the actual accesses. To build a useful map of memory accesses performed by a parallel application, events in a memory trace should also include information about time, space (at which address the event occurs), location (on which CPU it occurs) and nature of access (is this a read, a write, by which thread). Furthermore, the trace has to be sufficiently precise, that is include a sufficient number of events, in order to enable a sound analysis. At the time of this writing, existing memory analysis tools are not able to provide such traces. Some of them rely on instructions sampling [5, 6] mechanisms, which produce incomplete traces at a precision limited by hardware capabilities. Other ones ignore temporal information to reduce their overhead [7].

Indeed, generating such traces is a challenge: there is no hardware mechanism comparable to CPU performance counters to collect a detailed trace of memory accesses, and the volume of data to collect is huge. Methods based on binary instrumentation are too slow to provide a *complete* trace, and efficient hardware sampling mechanisms are not designed to provide all the information that constitute a *complete* trace.

In this study, we present *Memory Organisation Cartography and Analysis* [1] an efficient memory trace collection system based on two existing techniques: page fault interception and false page fault injection. This tool is able to collect a *complete* sampled memory trace at the spatial granularity of the page along with a parametrized temporal granularity. It also collects detailed information about sampled accesses, including their address and precise timestamp. Furthermore, *Moca* is able to

[1] *Moca* is distributed under GPL licence: github.com/dbeniamine/MOCA

retrieve data structures information (address, size and name) using the informations contained in the application binary.

The remaining of this paper is organised as follows: in section 2 we discuss about related works, in section 3 we present *Moca* design, then, we evaluate *Moca* by comparing it to existing tools in section 4. Finally, we present our conclusions and some possible future work in section 5.

## 2 Related Works

Several generic tools have been designed to analyze and improve parallel applications performances, such as Intel's VTune [3], Performance Counter Monitor (PCM) [8], the HPC-Toolkit [4], and AMD's CodeAnalyst [9]. All of these tools use performance counters and execution traces to show when and where CPUs are idle, and, thus, highlight potential places for improvements. But they mostly focus on CPU related information and provide only indirect data about memory performances, ones that can be computed from the performance counters values.

As performances counters are architecture dependent and are not always easily understandable, higher level libraries such as *PAPI* [10] and *Likwid* [11] have been developed to ease their analysis. These libraries are able to derive more abstract and understandable metrics from raw hardware counters. For instance Likwid provides several groups of metrics related to memory providing the user with computed bandwidth in each cache level and between each CPU core and the main memory. Overall, a lot of studies provide a memory analysis solely based on information collected through these hardware performance counters [12, 13, 14, 15, 16, 17]. Despite providing a good global summary about memory performance figures, these counters only provide a partial view of the execution. They account for memory related events from the point of view of one processor, but do not give a precise insight about the place and the cause of these events.

Another approach used by several tools [6, 18, 5, 19] consists in using instructions sampling mechanisms such as AMD's Instruction Based Sampling (IBS) [20] or Intel Precise Event Based Sampling (PEBS) [21] to trace the application execution. These methods provide *incomplete* sampling: some parts of the memory can be accessed without being noticed by the tool if none of the associated instructions are part of the sampled instructions. Thus, it is possible that they ignore some parts of the memory less frequently accessed but in which optimization could take place. Despite their low frequency, application sensitive to spurious performance degradation, such as interactive applications, could be hindered by these unnoticed accesses.

To make things practical, these sampling mechanisms monitor what they name an events set given by an instruction type along with some predicates. They can monitor several events sets at the same time but the number of monitored sets is limited by the hardware capabilities (number of available registers). Unfortunately, the number of existing events sets that relate to the memory hierarchy is large, because of its complexity. This makes difficult the task of tracing all the relevant memory accesses with just a single analysis. One way to lessen the impact of this limitation is to run several times the instrumentation and use advanced methods such as folding [22] to generate a more accurate summary trace. Nevertheless, this makes the instrumentation cost grow accordingly. Moreover, writing (and sometimes) using tools that relies on hardware mechanisms requires a deep knowledge of the processor. As processors evolve, such tools are hard to maintain and can quickly become outdated. We regard all these limitations as too constraining for a general purpose memory analysis tool.

Some other studies make use of hardware modification, either actual or simulated [23, 24]. Although they are eventually able to collect more precise traces efficiently, these techniques are limited to hardware developers. Indeed, to use these hardware extensions one has either to obtain (or build) a prototype or to use a suitable simulator. Such configuration is not realistic for general purpose memory analysis.

Binary instrumentation can provide accurate information about memory accesses. This method is portable and more precise than the aforementioned ones, but it comes at the cost of performances. Tools that rely on binary instrumentation usually collect data at a coarser granularity and give up temporal data [25] to reduce their overhead.

Page faults interception can provide useful online information about memory usage, such a mechanism has been used in several existing works : in parallel garbage collectors [26], in memory checkpointing [27] or in the domain of virtualization to provide the hypervisor with information about the memory usage of the guest OS [28]. Nevertheless, page faults only occur when caused by predetermined events in the system. Thus, just intercepting existing page faults only provide a broad view of the memory behavior. To reach a deeper understanding of this memory behavior, it is also possible to fake invalid pages at regular intervals in order to generate false faults [29, 30]. These false page faults are just triggered during regular memory accesses that would not have caused a page fault if the page were not faked as invalid. The advantage is that they create additional events for the monitoring tool to collect, thus more precision, but the set of faked invalid pages has to be known and maintained by the monitoring tool.

As a final note, most tools close to our proposal do not use false page faults injection and only need to store the location of memory pages and the threads that access them. As a consequence, they require a relatively small data structure in memory for their own usage. In this study we present *Moca*, a new *complete* memory trace collection system, based on page fault interception and false page faults injection, able to capture precisely the temporal evolution of multithreaded applications memory accesses. To reach a satisfying precision, our tool has to maintain in memory both the trace data and the set of faked invalid pages. Overall, storing and exploiting efficiently these data within the kernel space and outputting it in real time to the user space is a challenge and is the main contribution of our work.

# 3 Design

*Moca* consist of a Linux kernel module that can be loaded at runtime, a script in charge of both loading this module with the proper parameters and launching the monitored application on the user behalf and an optional *Pintool* [31] (c++ library based on Intel Pin) to retrieve data structure informations. It neither relies on architecture specific technologies such as AMD IBS or Intel PEBS, nor on architecture dependent kernel code, kernel patch or kernel modifications. Therefore it is highly portable and can be run on any recent Linux kernel from the 3.0.

Two tasks are addressed by the kernel module included in *Moca*. The first one is keeping track of the set of pages accessed by the application during an elementary monitoring interval. The second one is managing somehow the huge quantity of data produced by the trace collection within the kernel space in-between regular flushes toward the user space. Of course, these two tasks should be as slightly intrusive as possible.

When the Pintool is enabled *Moca* runs the application twice with `virtual address space randomization` disabled. The first time the application runs only under Pin instrumentation and the second time *Moca* module is loaded but the instrumentation is disabled. This instrumentation reads static data structures information in each executed binary file and stores data about structures larger than one page. It also intercepts all calls to `malloc` family functions and names malloced structures according to their call path. *Moca* and Pin are run separately because preliminary experiments showed us that combining both tools resulted in a degradation of the trace quality. Our instrumentation remains lightweight, the added cost of the Pin instrumentation is the cost of a regular execution added to a small constant overhead for each binary opened and each allocation performed.

### 3.1 Collecting memory accesses

*Moca* collects *complete* traces in the sense that the exact set of pages accessed by the application is deduced from the collected events at all times during the execution. Thus, it is *complete* at the page granularity. Other information such as exact addresses and access times are a sample of the set of all the accesses.

In recent Linux kernel, physical memory pages are lazily allocated to page frames during the execution. The first access to a page in the virtual address space triggers a page fault. To handle this page fault, Linux allocates a physical page to the requested page frame. Such a page fault can also be triggered when a thread access a shared page modified by another thread. *Moca* is built upon the possibility to monitor memory accesses by registering a callback on Linux page faults.

Nevertheless, a page fault does not occur at each memory access. To monitor memory accesses during the course of the execution, we need to reenable a page fault similar to the first access, but performed on a regular basis and on behalf of *Moca*. In other words, we need to generate false page fault by periodically marking as *not present* the pages accesses by the application. In Linux terminology, this means that any access to the page will trigger a page fault which will have to be handled, in this case, by a handler contained in *Moca*.

This method has several advantages over hardware sampling or instrumentation. First it provides a superset of all the memory accesses, because it guarantee that each page accessed by the monitored application will at least fault once and will be traced. Thus, at the end of each monitoring interval, we know the exact set of accessed pages from which we deduce a superset of actual memory accesses. This comes in addition to the fact that each false page fault generated provides *Moca* with exact information about one memory access. This means that *Moca* also performs a sampling of all the memory accesses. Because it is designed to manage large chunks of trace data within the kernel space, it also stores all the details about these samples in the collected trace.

*Moca* differs from instruction sampling because it is not necessary to increase the monitoring frequency of *Moca* to collect a *complete* trace. On the contrary, when using instruction sampling, if the pages of the application are accessed in an unbalanced manner, it is necessary to increase the sampling frequency to get a precise picture of the memory working set of the application. Nevertheless, there can be no guarantee that a chosen sampling frequency will result in a trace that contains all the pages on which the application works.

*Moca* also differs from instrumentation based tools because, just as in the case of sampling, memory accesses that are not collected in the trace are not trapped at all by a false page fault. Furthermore, the remaining memory accesses, those which are collected, are trapped using a hardware mechanism and Linux kernel probes. Both are lightweight mechanisms, this means that the overall instrumentation overhead of *Moca* is likely to be much lower. Indeed, instrumentation based methods often work at a high granularity, collecting few information, in order to restrain their naturally high overhead.

### 3.2 Managing data

In this section we present in details how the main components of *Moca* interact and how *Moca* addresses the management of data it collects within the kernel space. During the execution, *Moca* needs to store three kinds of information:

1. The set of *tasks* (Linux internal representation of threads and processes) which are monitored. This is necessary because page faults will also be triggered by other tasks which do not belong to the monitored application.

2. The set of all page faults which have been injected by *Moca*, required to distinguish false page faults from regular ones, because their handling differs.

3. The set of addresses recently accessed by each task, this set correspond to the actual memory trace. It is required to keep

it in kernel space as we need to reinject these false page faults at the end of each monitoring interval. Afterwards, this set is transfered to the user space by a dedicated process and appended to the resulting trace.

The first two types of information are stored in preallocated hashmaps in order to reduce the runtime overhead of their management. These hashmaps are read at each page fault but rarely written, only when a new task of the monitored application triggers its first page fault or when *Moca* creates some false page faults. We can protect them with Linux kernel built-in *rwlocks*. The third type of information is the actual trace, divided, for each task, in a private set of *chunks*. A chunk is the set of accesses that have been collected during the monitoring time interval. Chunks provide a discretization of the time, each chunk embed two timestamp to delimit its temporal bounds. The accesses are not timestamped but their order in the trace file is their order of arrival in the chunk. The advantage is that it reduces the volume of information stored for the accesses.

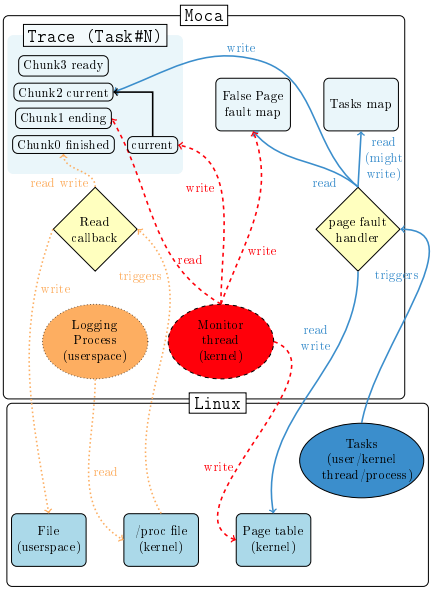



Figure 1: Interactions betweens *Moca* and Linux

This discretization of the time, materialized as a sequence of chunks, is useful as it let the different components of *Moca* work concurrently on different chunks. Indeed the traced program will always work on *current* chunks, one for each core, while the logging daemon, which flushes the trace from memory to permanent storage works on *completed* chunks. A monitoring kernel thread, manages the progress of this logical time. It periodically wakes up, marks the current chunks as *ending* and invalidates all the pages they reference. Once all pages of the *ending* chunks have been invalidated, it marks these chunks as *completed*. Finally, the logging process flushes *completed* chunks to the filesystem at a lower rate, in order to reduce the overhead of I/Os requests, and recycle them as empty places for upcoming chunks. Figure 1 depicts the interaction between the different processes and threads of *Moca*, its data structures and Linux.

Each time a page fault occurs, it is trapped by the handler registered by *Moca*, which first finds out if the task (thread or process) responsible for the page fault is monitored or not. If not, it has to check if the task is a child of a monitored task and, in this case, it starts monitoring it. This can be done with just a read lock on the hashmap containing the monitored tasks. The write lock is only taken if the task must be added. This case occurs only at the first page fault of a new monitored process or thread which is quite rare and usually occurs only at the initialization time. For instance, in the benchmarks used for the evaluation it happens 8 times out of $5 \times 10^6$ accesses. At the end of this phase, if the task is still not monitored, we let Linux handle the page fault as usual.

When a monitored task triggers a page fault, the access is first added to its current chunk. For each access, *Moca* stores the exact address, its type (read or write) and the CPU on which the fault occurred. Then, it checks if the page fault has been injected by *Moca* or if this is a legitimate page fault. In the first case, *Moca* *fixes* it by setting the `PRESENT` flag on the `Page Table Entry`. The hashmap entry indicating that the fault was triggered by *Moca* should then be removed, but this would required a write lock, so we only mark the hashmap en-

try as `BAD`. `BAD` entries are removed, if needed, when the monitoring thread injects false page faults as this injection already requires to hold the write lock. If a fix occurred in the *Moca* handler, Linux silently aborts the page fault when it resumes its execution. In the other case, it executes a normal page fault handling. Each page fault increases an atomic clock that is used to timestamp the beginning and end of the chunks.

| | | | | | |
|---|---|---|---|---|---|
| CPU | `Edel` | Intel Xeon E5520 | | | |
| | `St Remi` | AMD Opteron 6164 HE | | | |
| System totals | | Nodes | Threads | Freq | Memory |
| | `Edel` | 2 | 8 | 2.27 Ghz | 24 Gib |
| | `St Remi` | 2 | 24 | 1.7 Ghz | 48 Gib |
| Per node | | Cores | Threads | L3 Cache | Memory |
| | `Edel` | 4 | 4 | 8 Mib | 12 Gib |
| | `St Remi` | 6 | 6 | 12 Mib | 24 Gib |

Table 1: Hardware configuration of our evaluation system.

| | Mechanisms | | | Architecture | |
|---|---|---|---|---|---|
| *Tabarnac* | Instrumentation | | | Intel, AMD | |
| *Mitos* | PEBS + Instrumentation | | | Intel | |
| *MemProf* | IBS | | | AMD | |
| *Moca* | Page faults (+Instrumentation) | | | **Any** | |
| | Granularity | Superset | Time | Thread sharing | CPU |
| *Tabarnac* | Page | **Page** | no | **yes** | no |
| *Mitos* | **Address** | None | **yes** | no | **yes** |
| *MemProf* | **Address** | None | **yes** | **yes** | **yes** |
| *Moca* | **Address** | **Page** | **yes** | **yes** | **yes** |

Table 2: Comparison of different memory traces tools.

# 4 Experiments

In this part, we compare *Moca* to other existing tools for memory analysis. In a first time, we present their main differences in terms of portability and capabilities. Then, we present two sequences of quantitative experiments, one that outlines the importance of the default parameters chosen for our tool and the other that compares the precision and performance of all the tools.

## 4.1 Methodology

Our main experiments were run on machines from Grid5000 `Edel` cluster. As some state of the art tools can only run on AMD machines, we also ran some of the experiment presented in section 4.3 on `St Remi` machine from Grid5000 grenoble. These machines hardware specifications [2] are summarized in Table 1.

For each experiment, we deployed the same *Debian Jessie* environment running a `Linux 3.16.0-4` on a machine with hyper threading disabled. We disabled address space randomization to make the comparison between different traces more practical. As our two evaluation machines do not have the same hardware, we limited the number of threads used by openMP to 8 that is the largest number of hardware threads available on both machines.

We evaluate *Moca* by comparing it to the following state of the art tools. The first one, *Mitos*, is the tracing tool from MemAxes [19]. The second one, *Tabarnac* [7], is one of our previous contribution, which only counts the number of time each thread accesses to each page. The third one, *MemProf* [6], is designed to analyze NUMA performance issues. The main differences between *Moca* and these other memory profiling tools are summarized in Table 2.

In the following sections, all the tools are evaluated on each of the 10 *NAS parallel benchmarks* [32], which are presented in Table 3, according to the information available on the nasa website[3].

Except for the experiment about the influence of *Moca*'s parameters, on each experiment, *Moca* has been run with it's default parameters: a wakeup interval of 0.5 s for the logging process and 50 ms for the monitoring thread. Each point in each plot is the average of at least 30 executions. Along with each point, the error bars represent the standard error.

We consider experimental reproducibility as

[2] Grid5000 provides an online hardware description: https://www.grid5000.fr/mediawiki/index.php/Grenoble:Hardware#Edel https://www.grid5000.fr/mediawiki/index.php/Reims:Hardware#Stremi

[3] http://www.nas.nasa.gov/publications/npb.html

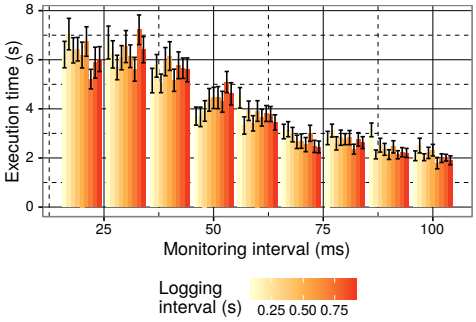
| Group | Name | Footprints* | Description |
|---|---|---|---|
| Memory Intensive | IS | 132 Mib | Integer Sort |
| | CG | 125Mib | Conjugate Gradient |
| | MG | 508Mib | Multi-grid |
| | FT | 398Mib | Discrete 3D FFT |
| Unstructured | UA | 112Mib | Unstructured Adaptive mesh |
| | DC | 1.46Gib | Data Cube |
| Pseudo Applications (solvers) | BT | 120Mib | Block Tri-diagonal |
| | SP | 122Mib | Scalar Penta-diagonal |
| | LU | 118Mib | Lower-Upper Gauss-Seidel |
| CPU bound | EP | 78Mib | Embarrassingly parallel |

Table 3: Description of the *NAS parallel benchmarks*.
**maximum memory used, measured with Valgrind.*

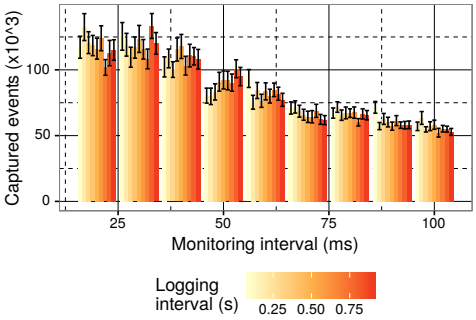
an important matter, therefore we distribute[4] all the files needed to reproduce our experiments at three different levels: The first level constains the filtered results (csv files) from the experiments along with the `R-markdown` scripts that generated the plots presented in this article. The second consists of the full raw traces generated by our experiments along with the scripts used to extract the filtered traces (csv files from the previous level) and the scripts used at the previous level to perform the analysis. Finally, at the most comprehensive level, we provide a git repository that includes our deployment environment, dependencies to all the tools and files required and instructions that explain how to reproduce the experiment with or without access to grid5000.

## 4.2 Moca default parameters

Before comparing *Moca* to existing tools, we need to evaluate the impact of the wakeup intervals (logging daemon and monitor thread) on the trace precision and on the overhead. To do so, we run the `IS` benchmark instrumented by *Moca* with a wakeup interval ranging from 0.1 s to 0.9 s for the logging daemon and from 20 ms to 100 ms for the monitoring thread. For each run, we measure `IS` execution time and the number of accesses captured. We have chosen `IS` for this evaluation as it is one of the memory intensive *NAS parallel benchmarks*, quick experiments with other ones confirmed these results. This experiment was run on a machine from the `Edel` cluster.




(a) Execution time.




(b) Number of captured events.

Figure 2: Influence of the wakeup intervals on `IS`, class A.

We can see on the Figure 2a that the execution time increases when we reduce the monitoring wakeup interval. At 40 ms it seems to reach its worst level, thus we should keep it larger. At 50 ms, the default value we have chosen, the Figure 2b shows that we obtain more than two thirds of the events captured at smaller intervals, which seems quite reasonable. Regarding the logging interval, our experiments do not exhibit a clear trend. Changing it seems to interfere with the system I/Os scheduler resulting in chaotic variations both in

[4] See our experiment repository: github.com/dbeniamine/Moca_expe

the execution time and the number of captured events. The fact that variations in execution time result in matching variations in the number of captured events is due to the fixed length of monitoring intervals : the longer the execution, the more monitoring intervals there are and the more events the trace contains. Overall these variations are not significant as all the confidence intervals intersect. Finally we have chosen a logging interval of 0.5 s, the median value, in order to avoid unnoticed effect caused by extremum values.

### 4.3 Comparison with existing tools

Preliminary experiments showed us that *Mitos* capture by default way less distinct pages than *Tabarnac* and *Moca*. Thus, we tried to change *Mitos* sampling period in order to make it capture at most pages as possible, we name this version *MitosTun*. Surprisingly, its behavior regarding this sampling period is not monotonous, we had to try many different periods to find the proper one.

The default *MemProf* distribution did not work with our experimental setup. With the help of their support team[5], we managed to make it work by disabling the library used to retrieve data structures names. For the same reason as in the case of *Mitos*, our study includes two version of *MemProf*: the default version and *MemProfTun* for which we have increased the sampling rate to its maximum.

Finally our evaluation also differentiate *Moca* (kernel module only) from *MocaPin* which also retrieve the data structure information using a Pin instrumentation, we make this distinction to evaluate the impact of Pin on *Moca* performances.

We compare the different tools on two aspects: trace precision and induced slowdown. Regarding the trace precision, the first experiment compares the tools using two criteria: the percentage of captured pages and the number of captured events. We use *Tabarnac* as a reference to compute the total number of pages accessed by the application because, by design, it traps all the memory accesses to compute the number performed in each page. This metric is representative of coverage of the memory space: the capacity of the tool to outline the whole memory area accessed by the application. Regarding the number of captured events, we present the percentage relative to *Moca*, as it is the tool that usually provides the more precise traces. We define one event as one timestamped access found in the trace file outputted by a tool. According to this definition, *Tabarnac* does not capture any events as it only keep one counter per page and per thread without any temporal informations. Thus, *Tabarnac* is excluded from this comparison. The number of captured events is representative of the precision of a monitoring tool, its capacity to keep track of all the evolutions of the access patterns during the course of the execution. The idea is that, the more the tool captures events, the less it misses changes in the access patterns.

The second experiment compares the slowdown factor of the different tools. All these experiments have been run on each of the *NAS parallel benchmarks* on class A.

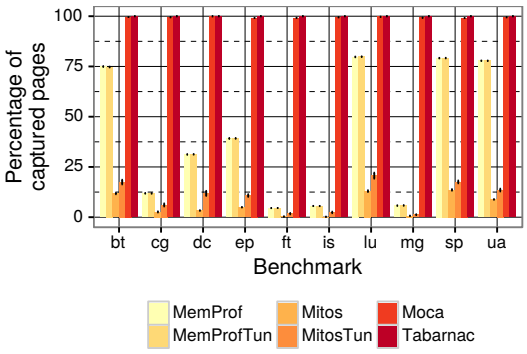
Figure 3 presents the results of the precision evaluation of the different tools. The values used for *Mitos*, *MitosTun*, *Moca* and *Tabarnac* result from runs on `Edel` machines, while *MemProf* and *MemProfTun* values result from runs on `St Remi`.

We can see on Figure 3a that *Moca* captures almost as many pages as *Tabarnac*. Regarding their design they should capture as many pages. Nevertheless, there is a slight bump in the number of pages used by applications monitored by *Tabarnac* due to the Pin instrumentation. Indeed, its JIT instrumentation recompiles the executable on the fly and changes the memory footprint (of the stack, mainly). Thus, we can safely ignore these differences.

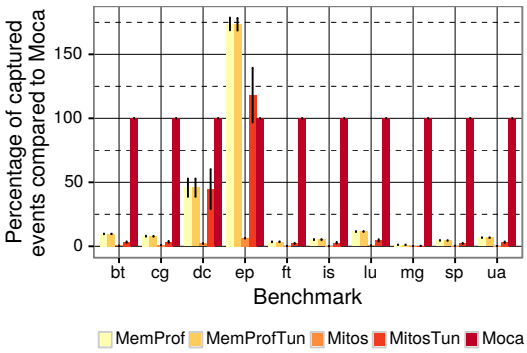
*Mitos* usually collect less than 12.5 % of the pages, adding some fine tunning can almost double this number but it still misses most of the address space.

Concerning *MemProf*, changing the default sampling rate does not seems to have any noticeable impact on the end result. Both *MemProf* and *MemProfTun* captures significantly

[5] see issue at github.com/Memprof/scripts/issues/1




(a) Percentage of captured pages.




(b) Percentages of events captured (compared to *Moca*).

Figure 3: Precision of the traces generated by each tool.

more pages than *Mitos* and *MitosTun*. Nevertheless, for half of the studied applications it does not see more than 50 % of the addresses space. Only for, `BT`, `LU`, `SP` and `UA`, *MemProf* manages to capture around 75 % of the accessed pages. This is explained by the fact that all these benchmarks are using uniformly most of their address space, and that many pages are frequently accessed. This is coherent with the fact that *MemProf* is solely based on instructions sampling and only sees the most accesses pages.

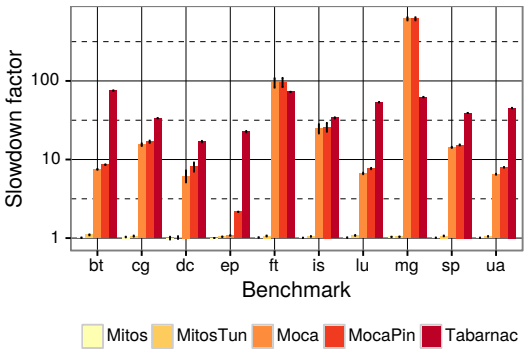
From Figure 3b we can see that, as expected, for almost every benchmarks, *Moca* collects significantly more events than the other tools. The only benchmark for which *Moca* is not the more precise tool is `EP` which is an Embarrassingly Parallel application with very few memory accesses. This outlines the fact that *Moca* captures events in an uniform way, timed by the monitoring interval. On the contrary, the other tools might capture more events in a few hotspots presents in the application but miss sparse accesses during the rest of the execution. For almost every other benchmarks both *Mitos* (with or without tunning) and *MemProf* hardly reach 10 % of the accesses collected by *Moca*, the only exception is `DC` for which *MemProf* captures from 25 % to 50 % of the accesses collected by *Moca*.

These results prove that most existing tools can miss a considerable part of the address-space while *Moca* guarantee that it provides a superset of the accessed pages. Furthermore they show that *Moca* is the only existing tool able to provide a trace that is precise enough to give an good overview of the memory behavior of an application. In short, not only our tool provides a complete trace at the granularity of the page but it is also significantly more precise than the other existing tools.

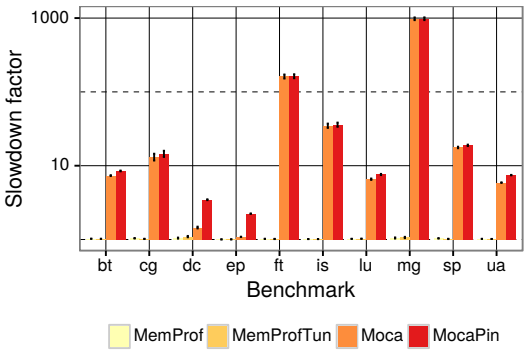
Figure 4 shows for each of the *NAS parallel benchmarks*, the slowdown factor when instrumented by *Moca* and the other existing tools on Intel (Figure 4a) and AMD (Figure 4b) Machines. Notice that the Y-axis is in log scale.

From Figure 4a, we can see that *Mitos*, *MitosTun* overhead is almost negligible which is not the case for *Moca* and *Tabarnac*, this difference is explained by the results of the previous experiment, as these tools usually collects less than 10 % of the accesses collected by *Moca* and miss a significant part of the address space.

We can classify the benchmarks into three groups: for `BT`, `CG`, `DC`, `EP`, `LU`, `SP` and `UA`, *Moca* is significantly faster than *Tabarnac*. This set of benchmarks is interesting as it is made of varied application profiles as we can see in Table 3. Indeed, if `EP` is mostly doing parallel computation with only a few number of memory accesses, `CG` is described as memory intensive, `BT`, `LU` as well as `SP` are linear algebra solvers with regular memory access patterns, and both `UA` and `DC` contain *unstructured computation, parallel I/O and data movement*. Furthermore, `DC` has




(a) Evaluation on `Edel` (Intel)




(b) Evaluation on `St Remi` (AMD)

Figure 4: Slowdown factor of each tool. Y-axis in log scale.

a considerable memory footprint as described in Table 3.

The second group only contains memory intensive benchmarks (`FT` and `IS`). For this group, *Moca* is as good as *Tabarnac* or a bit faster, probably because the balance between computations and memory accesses hides the overhead of the instrumentation.

For the last benchmark: `MG`, *Moca* is significantly slower than *Tabarnac*. By looking at our experiment logs, we found that `MG` generates a lot of conflicts in the hash map used by *Moca* to store false page faults. This issue is caused by applications that perform a very large number of sparse accesses to a large working set. This is not usual as parallel applications are often optimized to make memory accesses as local as possible in order to take advantage of all the levels of the memory hierarchy. Thus, we consider this benchmark as a pathological case. A solution could be to increase the size of this hash map, which is not always possible as memory space in the kernel is limited (and these experiments have been run with the largest hash map we could use). Another easier solution would consist in working on a smaller instance of `MG` and see if the trace is still useful. Although the results are not presented here, we have run *Moca* on `MG` with a smaller size (W) and we have been able to confirm that the performance becomes comparable to *Tabarnac* in this case.

Figure 4b shows the results of the evaluation on the AMD machine (`St Remi`). On this machine, *Moca* overhead is quite similar to the one obtained on `Edel`. *MemProf* exhibits a slowdown factor comparable to *Mitos* while providing traces a little more precise. Nevertheless, they are still *incomplete* and way less precise than *Moca* traces. Obviously *MemProfTun* has the same overhead as *MemProf* as it capture the same amount of data.

Finally, we can see, as expected, that adding one execution with a Pin instrumentation to retrieve data structures information (*MocaPin*) only adds a small overhead to the whole *Moca* execution. For several benchmarks this difference is so small that we cannot distinguish it from *Moca* usual overhead.

# 5 Conclusions

In this study, we addressed the issue of memory accesses collection for multithreaded applications. This is a key challenge in high performance computing as memory is often a performance bottleneck. Memory traces can be used at runtime to improve data locality or offline by developers to understand and improve the memory behavior of their applications and, therefore, their performances. For online analysis the trace precision is limited by the number of data that can be analyzed in real time, but for offline usage, highly accurate traces can provides a better understanding of the application

memory behavior.

To address this challenge, we have proposed *Moca* an efficient tool for precise and complete memory trace collection. While other existing tools rely on *incomplete* hardware sampling to provide such traces at an acceptable cost, *Moca* provides a *complete* trace, that contains all the accessed areas, at the granularity of the page. Moreover, *Moca* traces not only contain all pages that are accessed during the execution, but also, for each access, they contain temporal, spacial and sharing information: which thread, accessed what addresses on which CPU and when. While *Moca* works at the page granularity, it stores the exact address of each intercepted accesses. Therefore, it also provide an *incomplete* trace at the granularity of the byte, similar to traces collected by instructions sampling. Furthermore *Moca* can also relate accesses to data structures of the application by combining this efficient trace collection system with an examination of the application binary.

Most state of the art tools are relying on hardware technologies such as Intel PEBS or AMD IBS, and embed vendor (or processor) dependent code making them hard to maintain and not portable. On the contrary, *Moca* is based on page fault interception as well as false page faults injection mechanisms and does not use any architecture dependent code. It can work on any Linux kernel from 3.0 only by loading a module and without any kernel modification.

Several tools uses page fault interception to retrieve information about the memory behavior. As the information provided by only intercepting regular page fault is not always precise enough, a few tools also inject false page fault on a regular basis to increase the trace precision. To our knowledge, all the existing tools relying on these mechanisms uses the collected data online and thus does not have to manage and store a large amount of data. *Moca* is the first tool capable of generating and storing *complete* and precise memory traces for offline analysis.

We evaluated *Moca* by comparing it to two state of the art tools: *Mitos* and *MemProf* (with their default parameters and with some fine tunning) and one tool from a previous contribution *Tabarnac*. For this comparison, we evaluated two criteria: the precision of the trace and the overhead. We ran our evaluation on the *NAS parallel benchmarks* which is representative of multiple kinds of applications from simple kernels to realistic ones. Our evaluation has exposed the fact that the tools relying on hardware sampling miss a significant part of the address space. It has also shown that *Moca* is able to provide both a complete trace at the page granularity and a sampling at the byte granularity significantly more precise than the other tools. While generating comparable traces using *MemProf* or *Mitos* would require to sample all the memory instructions, which is not possible. Finally, *Moca* overhead appears to be more important than the overhead of sampling based tools but usually lower than the one induced by binary instrumentation.

Future work will focus on the visualization and exploitation of these memory traces which is another challenge mainly due to the volume of the collected data.

## Acknowledgements

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 s other funding bodies (see `https://www.grid5000.fr`).

## References

[1] U. Drepper, "What every programmer should know about memory," http://people.redhat.com/drepper/cpumemory.pdf, 2007.

[2] J. Corbet, "AutoNUMA: the other approach to NUMA scheduling," 2012. [Online]. Available: http://lwn.net/Articles/488709/

[3] J. Reinders, *VTune performance analyzer essentials*. Intel Press, 2005.

[4] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010. [Online]. Available: http://dx.doi.org/10.1002/cpe.1553

[5] X. Liu and J. Mellor-Crummey, "A Tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 259–272. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555271

[6] R. Lachaize, B. Lepers, and V. Quema, "MemProf: A Memory Profiler for NUMA Multicore Systems," in *USENIX 2012 Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012, pp. 53–64. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/lachaize

[7] D. Beniamine, M. Diener, G. Huard, and P. O. A. Navaux, "TABARNAC: Tools for Analyzing Behavior of Applications Running on NUMA Architecture," Tech. Rep. 8774, Oct 2015. [Online]. Available: https://hal.inria.fr/hal-01202105

[8] Intel, "Intel Performance Counter Monitor - A better way to measure CPU utilization," 2012. [Online]. Available: http://www.intel.com/software/pcm

[9] P. J. Drongowski, "An introduction to analysis and optimization with AMD CodeAnalyst™ Performance Analyzer," Tech. Rep., 2008.

[10] V. M. Weaver, D. Terpstra, H. McCraw, M. Johnson, K. Kasichayanula, J. Ralph, J. Nelson, P. Mucci, T. Mohan, and S. Moore, "PAPI 5: Measuring power, energy, and the cloud," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, April 2013, pp. 124–125.

[11] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[12] Z. Majo and T. R. Gross, "(Mis)understanding the NUMA memory system performance of multithreaded workloads," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, Sept 2013, pp. 11–22.

[13] T. Jiang, Q. Zhang, R. Hou, L. Chai, S. A. Mckee, Z. Jia, and N. Sun, "Understanding the behavior of in-memory computing workloads," in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, Oct 2014, pp. 22–30.

[14] R. Bosch, C. Stolte, D. Tang, J. Gerth, M. Rosenblum, and P. Hanrahan, "Rivet: A Flexible Environment for Computer Systems Visualization," *SIGGRAPH Comput. Graph.*, vol. 34, no. 1, pp. 68–73, feb 2000. [Online]. Available: http://doi.acm.org/10.1145/563788.604455

[15] B. Weyers, C. Terboven, D. Schmidl, J. Herber, T. W. Kuhlen, M. S. Muller, and B. Hentschel, "Visualization of Memory Access Behavior on Hierarchical NUMA Architectures," in *Visual Performance Analysis (VPA), 2014 First Workshop on*, Nov 2014, pp. 42–49.

[16] J. Tao, W. Karl, and M. Schulz, "Visualizing the Memory Access Behavior of Shared Memory Applications on NUMA Architectures," in *Computational Science - ICCS 2001*, V. Alexandrov, J. Dongarra, B. Juliano, R. Renner, and C. J. K. Tan, Eds. Springer Berlin Heidelberg, 2001, vol. 2074, ch. Lecture Notes in Computer Science, pp. 861–870. [Online]. Available: http://dx.doi.org/10.1007/3-540-45718-6_91

[17] L. A. DeRose, "The Hardware Performance Monitor Toolkit," in *Euro-Par 2001 Parallel Processing*. Springer Berlin Heidelberg, 2001, vol. 2150, ch. Lecture Notes in Computer Science, pp. 122–132. [Online]. Available: http://dx.doi.org/10.1007/3-540-44681-8_19

[18] C. McCurdy and J. Vetter, "Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms," in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, March 2010, pp. 87–96.

[19] A. Giménez, T. Gamblin, B. Rountree, A. Bhatele, I. Jusufi, P.-T. Bremer, and B. Hamann, "Dissecting On-Node Memory Access Performance: A Semantic Approach," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 166–176. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.19

[20] P. J. Drongowski, "Instruction-based sampling: A new performance analysis technique for AMD family 10h processors," AMD CodeAnalyst Project, Boston Design center, Tech. Rep., November 2007.

[21] D. Levinthal, "Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors," Tech. Rep., 2009.

[22] H. Servat Gelabert, "Towards instantaneous performance analysis using coarse-grain sampled and instrumented data," Ph.D. dissertation, Universitat Politècnica de Catalunya, Barcelona, 2015.

[23] Y. Bao, M. Chen, Y. Ruan, L. Liu, J. Fan, Q. Yuan, B. Song, and J. Xu, "HMTT: A Platform Independent Full-system Memory Trace Monitoring System," in *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '08. New York, NY, USA: ACM, 2008, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/1375457.1375484

[24] M. Martonosi, A. Gupta, and T. Anderson, "MemSpy: Analyzing Memory System Bottlenecks in Programs," in *Proceedings of the 1992 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '92/PERFORMANCE '92. New York, NY, USA: ACM, 1992, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/133057.133079

[25] D. Beniamine, M. Diener, G. Huard, and P. O. A. Navaux, "TABARNAC: Visualizing and Resolving Memory Access Issues on NUMA Architectures," in *Proceedings of the 2Nd Workshop on Visual Performance Analysis*, ser. VPA '15. New York, NY, USA: ACM, 2015, pp. 1–1. [Online]. Available: http://doi.acm.org/10.1145/2835238.2835239

[26] H.-J. Boehm, A. J. Demers, and S. Shenker, "Mostly Parallel Garbage Collection," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, ser. PLDI '91. New York, NY, USA: ACM, 1991, pp. 157–164. [Online]. Available: http://doi.acm.org/10.1145/113445.113459

[27] J. Heo, S. Yi, Y. Cho, J. Hong, and S. Y. Shin, "Space-efficient Page-level Incremental Checkpointing," in *Proceedings of the 2005 ACM Symposium on Applied Computing*, ser. SAC '05. New York, NY, USA: ACM, 2005, pp. 1558–1562. [Online]. Available: http://doi.acm.org/10.1145/1066677.1067026

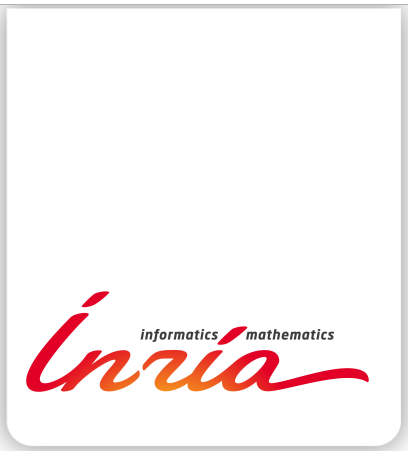
[28] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 14–24, oct 2006. [Online]. Available: http://doi.acm.org/10.1145/1168919.1168861

[29] C. S. Bae, L. Xia, P. Dinda, and J. Lange, "Dynamic Adaptive Virtual Core Mapping to Improve Power, Energy, and Performance in Multi-socket Multicores," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '12. New York, NY, USA: ACM, 2012, pp. 247–258. [Online]. Available: http://doi.acm.org/10.1145/2287076.2287114

[30] M. Diener, E. H. M. Cruz, and P. O. A. Navaux, "Communication-Based Mapping Using Shared Pages," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, May 2013, pp. 700–711.

[31] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[32] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS Parallel Benchmarks and Its Performance," NASA, Tech. Rep. October, 1999.




**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