TABARNAC: Visualizing and Resolving Memory Access Issues on NUMA Architectures

David Beniamine, Matthias Diener, Guillaume Huard, Philippe Olivier Alexandre Navaux

TABARNAC: Visualizing and Resolving Memory Access Issues on NUMA Architectures

David Beniamine
Univ. Grenoble Alpes, LIG,
F-38000 Grenoble, France
CNRS, LIG, F-38000
Grenoble, France
Inria
David.Beniamine@Imag.fr

Matthias Diener
Informatics Institute
UFRGS
Porto Alegre, Brazil
mdiener@inf.ufrgs.br

Guillaume Huard
Univ. Grenoble Alpes, LIG,
F-38000 Grenoble, France
CNRS, LIG, F-38000
Grenoble, France
Inria
Guillaume.Huard@Imag.fr

Philippe O. A. Navaux
Informatics Institute
UFRGS
Porto Alegre, Brazil
navaux@inf.ufrgs.br

ABSTRACT

In modern parallel architectures, memory accesses represent a common bottleneck. Thus, optimizing the way applications access the memory is an important way to improve performance and energy consumption. Memory accesses are even more important with NUMA machines, as the access time to data depends on its location in the memory. Many efforts were made to develop adaptive tools to improve memory accesses at the runtime by optimizing the mapping of data and threads to NUMA nodes. However, theses tools are not able to change the memory access pattern of the original application, therefore a code written without considering memory performance might not benefit from them. Moreover, automatic mapping tools take time to converge towards the best mapping, losing optimization opportunities. A deeper understanding of the memory behavior can help optimizing it, removing the need for runtime analysis.

In this paper, we present *TABARNAC*, a tool for analyzing the memory behavior of parallel applications with a focus on NUMA architectures. *TABARNAC* provides a new visualization of the memory access behavior, focusing on the distribution of accesses by thread and by structure. Such visualization allows the developer to easily understand why performance issues occur and how to fix them. Using *TABARNAC*, we explain why some applications do not benefit from data and thread mapping. Moreover, we propose several code modifications to improve the memory access behavior of several parallel applications.

VPA2015, November 15-20 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2835238.2835239 ...$15.00.

1. INTRODUCTION

Using memory on modern parallel shared-memory systems with a *Non-Uniform Memory Access (NUMA)* behavior is both trivial and extremely complex: an application is able to access the whole memory with the same interface, but to use it efficiently, the developer needs to take several performance factors into account, such as the cache hierarchy and the structure of the NUMA architecture [14]. NUMA machines are characterized by multiple memory controllers per system [3], dividing the physical main memory into several NUMA nodes. Each node can access its local memory directly, but has to transfer data through an interconnection network to access memory on remote nodes. Current systems usually have one memory controller per socket, but architectures with multiple controllers per socket are becoming more common [2]. In NUMA systems, decisions about where to place the data that a parallel application uses have a significant impact on the overall performance, with most policies aiming at improving the *locality* of memory accesses [11].

The optimal mapping of memory pages to NUMA nodes depends on the way an application accesses the memory. To improve the mapping without changing the application, several automatic tools were proposed [7, 8, 12, 32]. However, these tools have a runtime overhead as they need to analyze the application behavior during execution and lose opportunities for improvements during this training. Furthermore, they are not able to change the memory access pattern for additional improvements. Therefore, if the memory behavior is not designed for NUMA machines, their improvements might be limited. For instance, if all threads are accessing data from a single memory page, remote memory accesses will be triggered from all NUMA nodes but one, wherever the page is mapped. This kind of issue can only be solved by modifying the memory access behavior in the source code of the application, requiring a deep understanding of its behavior.

Several tools, such as Intel's VTune [33] and Performance Counter Monitor (PCM) [19], the HPCToolkit [1], and AMD's CodeAnalyst [16], can be used to help the developer under-

stand and improve the performance of parallel applications. However, these tools rely on hardware performance counters and can therefore provide only indirect and sampled information about the memory access behavior, through cache miss statistics, for example. Indeed, tracing the memory behavior is complex, as many instructions trigger at least one memory access. Several studies have addressed this problem using sampling [23, 31, 18], and can find out *what* happens (remote accesses, cache misses ...), *where* (data structure, line of code), but not *how* data structures are accessed and shared by the different threads (which cause the remote accesses).

In this paper, we present *TABARNAC*[1], a set of *Tools for Analyzing the Behavior of Applications Running on NUMA ArChitectures*. *TABARNAC* provides tools to trace and visualize the memory access behavior of parallel applications. More precisely, it helps to understand *why* performance issues occur by providing information on how data structures are accessed and shared by the different threads. Since it is based on memory accesses traces, *TABARNAC* has a very high accuracy while maintaining a reasonable overhead that enables the analysis of large applications. In an evaluation with several parallel applications, we show that relatively small code changes suggested by *TABARNAC* can substantially improve application performance.

The rest of this paper is organized as follows. In the next section, we discuss related work and compare it to our proposal. Section 3 presents the design and implementation of *TABARNAC*. Our evaluation methodology is outlined in Section 4. We show example analyses and performance improvements with *TABARNAC* using several parallel applications in Section 5. Finally, we present our conclusions and discuss ideas for future work in Section 6.

2. RELATED WORK

This section presents an overview of related work in the area of memory access profiling for parallel applications based on shared memory. We also discuss some mechanisms to improve performance on NUMA architectures.

2.1 Memory Profiling

Generic tools to evaluate parallel application performance, such as Intel's VTune [33] and Performance Counter Monitor (PCM) [19], the HPCToolkit [1], and AMD's CodeAnalyst [16], provide only indirect information about the memory access behavior, more specific tools are therefore required to improve it.

Profiling memory behavior raise two major challenges. The first one is the collection of accurate and detailed information: performance counters provide precise and easy access to statistics about the CPU usage, but there are few such mechanisms for the memory. For a maximum level of detail, memory access traces need to be created. The second challenge is the amount of information that needs to be interpreted and presented to the developer. Memory access traces provide huge amounts of information on several dimensions: data structure, threads, access type (read/write), sharing, and time of access. Presenting them to the developer in a readable and meaningful way is therefore not trivial.

[1] *TABARNAC* is available at:
`https://github.com/dbeniamine/Tabarnac`

2.1.1 Data Collection

Several methods have been used to address the problem of data collection. A lot of studies deduce information from hardware performance counters [28, 20, 5, 36, 35, 10], which are special registers that allow to record events such as cache misses and remote memory accesses. However, these counters only provide a partial view of the execution, they show events happening on the processor related to memory, but not what triggered them. Moreover, most available performance counters depend on the architecture, therefore it is hard to reproduce the same analysis on different machines with these tools.

Another approach used by several tools [23, 31, 25, 18] consists of using sampling mechanisms such as AMD's Instruction Based Sampling (IBS) [15] or Intel Precise Event Based Sampling (PEBS) [24] to analyze applications. Not only can sampling miss important events, leading to inaccurate characterizations, but these technologies are usually not portable and work only with a few recent architecture, therefore such tools can only be used in special circumstances.

Other studies uses hardware modification (with or without simulation) [4, 30]. Although they provide more efficient trace collection than tools implemented purely in software, they are even less portable. Finally, binary instrumentation can provide information about memory access behavior [9], although this method is slower than the other previously described techniques, it is more portable and precise. Moreover, as we show in Section 5.3, an efficient instrumentation can provide an acceptable overhead.

2.1.2 Visualization

The second difficulty of memory analysis is to present the information in such a way that the developer can use it to improve the application. Some of the tools previously mentioned only provide a textual output [23, 31, 30]. Even if these tools highlight the most relevant informations, it is hard to get an overview of the memory behavior from such output. The developer might be presented with a huge amount of information and not be able to differentiate normal behaviors from problematic ones.

Other tools provide more advanced visualizations. For instance, Tao et al. [35] propose a detailed view of each memory page, showing the number of remote and local accesses from each NUMA node. Weyers et al. [36] depict the memory bandwidth between each pair of nodes, showing where the remote accesses occur. Other tools [10, 9, 5] provide several views of the execution, giving the ability to correlate them with the source code of applications, similar to traditional performance tools such as VTune. Although all these tools can help developers understand the kind of performance issues they are facing, they do not give the reason *why* a particular issue is happening, for instance by showing the distribution of memory accesses within data structures.

MemAxes [18] is one of the most advanced NUMA-oriented visualization tools. Figure 1 shows a screenshot of this tool on an example trace. It shows the source code of the application (left upper side), the NUMA hierarchy of the machine (right upper side) and a *parallel coordinate graph* (lower side) designed to help correlating information. Although this visualization is designed to help understanding NUMA performance issues, it shows *which* event occurs and *where* it occurs, but does not tell directly *why* it occurs. The user still has to correlate several pieces of information to guess

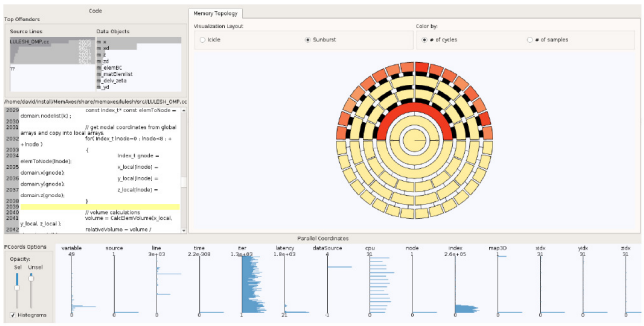


Figure 1: Screenshot from MemAxes on the example data trace provided with the tool.

the source of a performance issue.

Finally, the proposal of Liu et al. [25] is quite similar to the previous studies, but they also provide an *address centric* visualization, which shows how much each thread accesses a data structure. Such a visualization is a bit closer to providing the source of the performance issue, but it does not show how the accesses are distributed inside a structure, and how the structure is shared between the threads.

2.2 Data Mapping Mechanisms

On NUMA architectures, data mapping mechanisms have the goal of improving the locality and balance of memory access between NUMA nodes. Traditionally, operating systems have used the *first-touch* [29], *next-touch* [26] and *interleave* [22] policies to map memory pages to NUMA nodes. The first-touch policy, which is the default policy in most operating systems (such as Linux), allocates a page on the NUMA node that performs the first memory access to it. It requires the developer to take care of which thread accesses data first, as an incorrect first access can hurt performance. In next-touch [26], each page is periodically migrated to the NUMA node that performs the next access to a page. This technique is more flexible than first-touch, but can lead to excessive page migrations. The interleave policy (available in Linux via the `numactl` tool [22]) distributes memory pages cyclically among all NUMA nodes, to improve load balance among memory controllers, but it does not take any locality into account.

Newer developments in operating systems focus on refining the data mapping during the execution of parallel applications, using online profiling [12, 8]. Recent versions of the Linux kernel contain the NUMA Balancing technique [7], which uses page faults to determine if a page should be migrated to a different NUMA node. Other solutions improve the data mapping in the compiler, the runtime or at a library level. Piccoli et al. [32] propose a compiler extension that analyzes the memory accesses patterns of parallel loops and uses this information to migrate pages before executing the loop.

Libraries such as libnuma [22] and MAi [34] provide the ability to allocate data structures on a particular NUMA node, or with an interleave policy. These techniques can achieve large improvements, but require a deep understanding of the applications' memory behavior to use them efficiently. Our study provides tools to easily understand the memory behavior and therefore enable the developer to improve performance significantly.

2.3 Summary of Related Work

Several studies already provide tools to analyze memory accesses. These tools usually point out *which* performance issues occur (such as a high number of cache misses or accesses to remote NUMA nodes), sometimes *where* they occur (such as information about the structure, function, or line of code). Some tools helps to correlate these information to guess *why* an issue is happening. However, no tool directly provides the reasons *why* such issues occur and *how* to fix them. Two types of information can help answering this question: which thread is responsible for the first touch (as the default page mapping of most operating systems depends on it), and how different threads access data structures. This study presents *TABARNAC*, a set of tools to explain *why* performance issues related to memory occur and *how* they can be resolved.

3. TABARNAC

TABARNAC: Tools for Analyzing the Behavior of Applications Running on NUMA ArChitecture is divided into two parts: the instrumentation tool, which collects information about memory accesses, and the visualization, which presents a meaningful interpretation of the trace. In this section, we discuss the implementation of both parts.

3.1 Collecting Memory Access Information

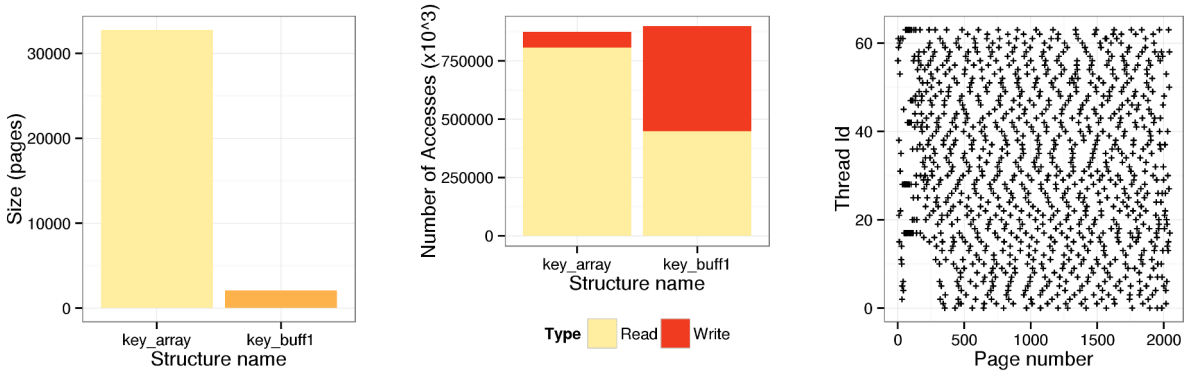
TABARNAC data collection aims at providing information on how data structure are accessed, therefore, it needs to collect fine-grained information. To do so, we instrument memory access and collect the number of access per page by thread and type (Read/Write). The information is stored on a per-thread basis, as shown in Listing 1, making the code completely lock-free, as well as minimizing the amount of false sharing between threads.

```
1  void mem_access(unsigned long address, int
       threadid, char type){
2    uint64_t page = address >> page_bits;
3    acc[threadid][page][type]++;
4  }
```

Listing 1: Code executed on each memory access. Pin provides the `address`, `threadid` and `type` parameters.

The instrumentation uses the Pin dynamic binary instrumentation tool [27]. Although it is an Intel technology, it works also on AMD processors. Previous versions of Pin also support Intel Itanium (IA64) and ARM architectures.

Before running the application, *TABARNAC* retrieves static memory allocation information. Dynamic allocations are intercepted at runtime and structure names are extracted using the debug information provided by the compiler. Finally, each time a thread is created, we compute its stack bounds and create a virtual structure named `Stack#N` where N is the thread ID. Only structures that are bigger than one page (usually 4Kib in current x86_64 architectures) are recorded as our analysis granularity is the memory page. The data structure information (name, size and address) are only used to generate the visualization, after the end of the instrumentation. The memory access tracing is based on the earlier numalize tool [13], which only collected statistics about memory accesses to pages, without information about data structure or stacks.



(a) Structures size. (b) Number of accesses per structures. (c) First touch distribution inside a structure.

Figure 2: Example plots from *TABARNAC*.

3.2 Visualization

Once the data collection phase is done, *TABARNAC* generates the visualization (as an HTML page), providing a summary of the trace through several plots[2]. The visualization aims at showing *why* performances issues related to memory occur, it therefore shows several plots helping to understand the importance of each data structure and how it is accessed. Each plot is introduced by an explanation of its presentation, what common issues it can help to understand and provides suggestions on how to fix these issues. The visualization starts with a small introduction, summarizing the main principles while developing for NUMA machines, and shows the hardware topology of the analyzed machine extracted with Hwloc [6].

After the introduction, the visualization focuses on the usage of data structures. Some structures are not displayed if less than 0.01% of the total accesses happen on them. This is done to make the output more readable by focusing on the most important structures.

The first series of plots presents information concerning the relative importance of the data structures. It consists of two plots, showing first the size of each data structure, as in Figure 2(a), then the number of reads and writes in each structure (Figure 2(b)). These plots give a general idea of the structures used by the parallel application. Moreover, knowing the read/write behavior is very useful as it determines the possible optimizations. For instance, structures written only during initialization (or very rarely) can be relatively easily duplicated, such that each NUMA node works on a local copy.

The second series of plots is the most important one. It shows for each page of each structure which thread was responsible for the first touch (Figure 2(c)). This information is important as the default policy for *Linux* and most other operating systems is to map a page as close as possible to the first thread accessing it. If the first touch distribution does not fit the actual access distribution, the default mapping performed by *Linux* might not be efficient. To address this issue, the developer can either correct the first touch or do some manual data mapping to ensure better memory access

[2]A full example of *TABARNAC*'s output is available at: `http://dbeniamine.github.io/Tabarnac/examples`.

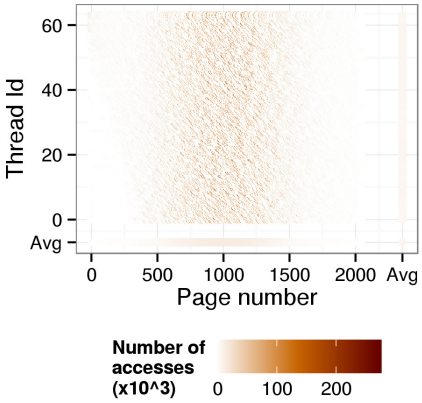


Figure 3: Per thread access distribution inside a structure.

locality and balance during the execution.

Finally, *TABARNAC* shows the density of accesses performed by each thread and the global distribution. In the example shown in Figure 3, each horizontal line represents the number of accesses to one page, there is one line per thread and one for the average number of accesses. Moreover, for each thread the average number of accesses to the structure is displayed. Darker lines indicate more memory accesses to the page. This visualization gives an easy way to understand the data sharing between threads, as well as the balance between pages and threads. These plots can be used to identify inefficient memory access behaviors and to determine the best NUMA mapping policy.

4. EXPERIMENTAL SETUP

This section briefly discusses our experimental setup for the evaluation of *TABARNAC*.

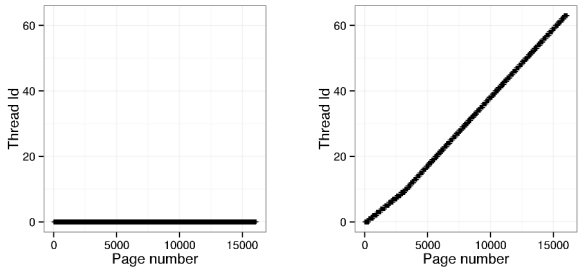
We used two NUMA machines for our experiments, `Turing` and `Idfreeze`. The second machine was only used to compare the instrumentation overhead on Intel and AMD machines, all the other experiments ran on `Turing`. The hardware details are summarized in Table 1. `Turing` runs ver-

CPU		Vendor	Model
	Turing	Intel	Xeon X7550
	Idfreeze	AMD	Opteron 6174

System totals		Nodes	Threads	Freq	Memory
	Turing	4	64	2.00 Ghz	128 Gib
	Idfreeze	8	48	2.20 Ghz	256 Gib

Per node		Cores	Threads	L3 Cache	Memory
	Turing	8	16	18 Mib	32 Gib
	Idfreeze	6	6	12 Mib	32 Gib

Table 1: Hardware configuration of our evaluation system.



(a) Original first-touch. (b) Improved first-touch.

Figure 4: First-touch for structure `vz0` from *Ondes3D*.

sion 3.13 of the Linux kernel, while `Idfreeze` runs version 3.2.

All applications use OpenMP for parallelization, they were compiled with `gcc`, version 4.6.3, with the `-O2` optimization flag. Both analysis and performance evaluation are performed with 64 threads, which is the maximum number of threads that our main evaluation machine (`Turing`) can execute in parallel.

In the performance evaluation, we compare the following three traditional mapping policies to the version modified using the knowledge provided by *TABARNAC*. The *original Linux kernel* is our baseline for the experiments. We use an unmodified Linux kernel, version 3.13, with the first-touch policy. The NUMA Balancing mechanism is disabled in this baseline. The *interleave* policy is performed with the help of the `numactl` tool [22]. We also compare our results to the recently introduced *NUMA Balancing* technique [7], which is executed with its default configuration.

For the plots presenting speedups, each configuration was executed at least 10 times. Each point shows the arithmetic mean of all runs. The error bars in those plots represent the standard error.

5. ANALYSIS AND RESULTS

This section presents the results of our analysis. For each application, we show its memory access behavior, discuss strategies to optimize this behavior and present the performance improvements that can be achieved.

5.1 Ondes3D

Ondes3D is the main numerical kernel of the Ondes3D application [17]. It simulates the propagation of seismic waves using a finite-differences numerical method. Ondes3D has a memory usage of 11.3Gib with the parameters used for the performance evaluation, and 0.7Gib for the analysis.

The analysis of the accesses distribution in *Ondes3D* (not displayed here) shows that each structure seems to be well distributed between the threads. However, for all structures, thread 0 is responsible for all first accesses, as we can see for `vz0` in Figure 4(a). Due to this pattern, if we run *Ondes3D* without any improved mapping policy, every page will be mapped to the NUMA node that executes the thread 0, resulting in mostly remote accesses for the other threads. An easy fix is to perform the initialization in parallel and to pin each thread on a different core, or to use the interleave policy. Such a modification results in the first touch distribution shown in Figure 4(b), which is now distributed among all the threads.

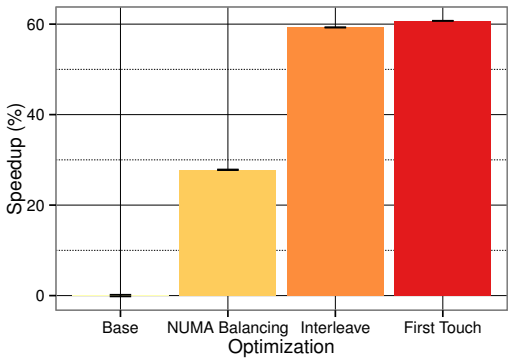
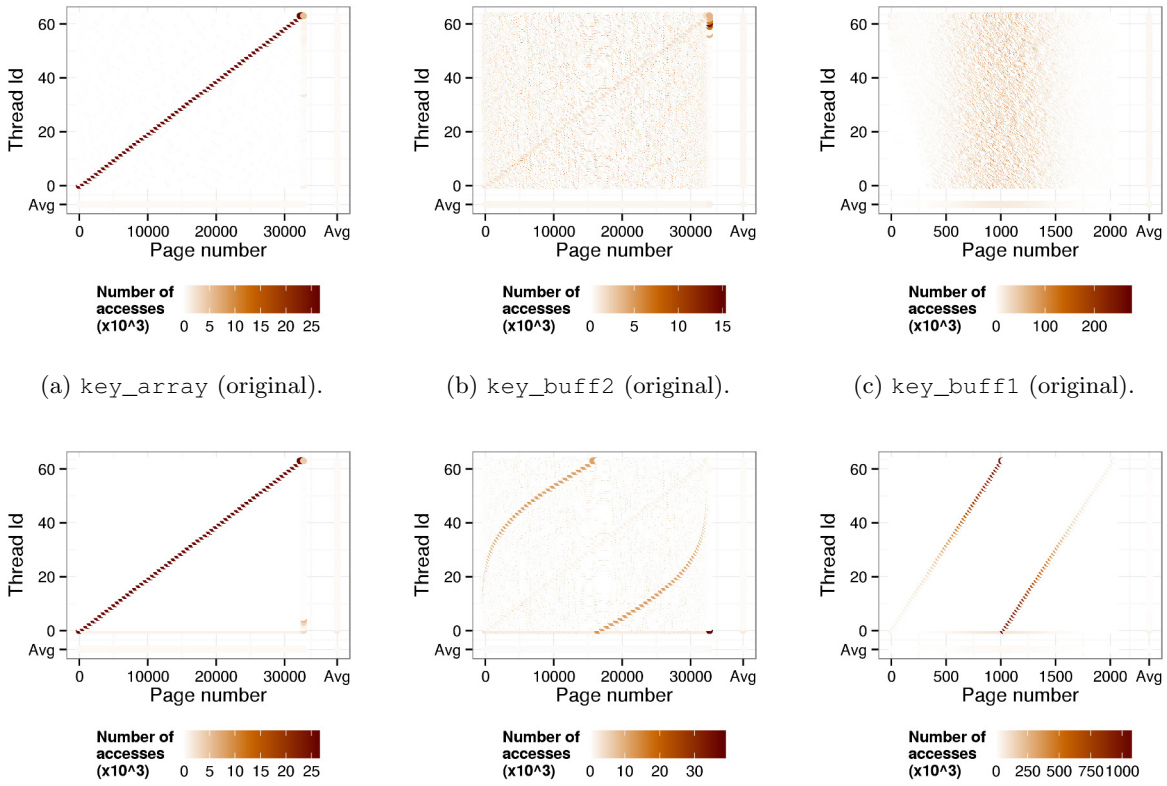


Figure 5: Speedup for *Ondes3D* compared to the baseline.

We compare the performance of the modified version (*First Touch*) to the original (*Base*) version, running on the normal OS, with *NUMA Balancing* activated and with an *Interleave* policy. Figure 5 present the results of this evaluation. We can see that all methods improve the execution time compared to the OS, but NUMA Balancing provides less than 30% speedup, while the static mappings (Interleave and the modified code) increase performance by 60%. Indeed, with *NUMA Balancing*, all pages are initially mapped by the OS to the NUMA node of thread 0, and are only moved later on, after many remote accesses have already occurred, losing some optimization opportunities. This is a case where static mapping can be substantially better than automated tools. The *Interleave* policy provides a similar speedup as *First Touch* since it distributes the pages over the NUMA nodes at the beginning of the execution, but our tool shows clearly the cause of the performances issue.

5.2 The IS Benchmark

We executed *TABARNAC* on the benchmarks from the OpenMP implementation of the NAS Parallel Benchmark suite (NPB) [21]. Most of them have either a well balanced accesses pattern between the threads or a totally random accesses distribution. For all of them, the first touch fits exactly the access distribution. However, the analysis of *IS* caught our attention. *IS* sorts a set of integer numbers using a parallel bucket sort algorithm. According to the



(a) `key_array` (original). (b) `key_buff2` (original). (c) `key_buff1` (original).

(d) `key_array` (modified). (e) `key_buff2` (modified). (f) `key_buff1` (modified).

Figure 6: Memory access distribution for the main structures of *IS*. Original behavior on the top, modified on the bottom.

NAS website[3], *IS* has a random memory access pattern, while we observed a very specific pattern. In this section we explain this pattern and how we used it to improve the performance of *IS*.

IS was executed with input class *D* for the performance evaluation, resulting in a memory usage of 33.5Gib, and class *B* for the analysis, with a memory usage of $0.25Gib$.

The top of Figure 6 shows the original access distributions for the three main structures of *IS*. We can see that each structure has a different access pattern: `key_array`'s (Figure 6(a)) access distribution shows that each thread works on a different part of the structure, which permits automated tools perform an efficient data/thread mapping on it. On the other hand, `key_buff2` (Figure 6(b)) is completely shared by all threads. `key_buff1`'s access distribution (Figure 6(c)) is the most interesting one. We can see that almost all accesses occur in pages in the middle of the structure (from page 500 to 1500), and those pages are shared by all threads. This means that the number of access per page for each thread follows a Gaussian distribution centered in the middle of the structure.

We can identify the source of this pattern in the *IS* source code. Indeed, all the accesses to `key_buff1` are linear, except in one OpenMP parallel loop where they depend on the value of `key_buff2`.

As we noticed in Figure 6(c) that the values of `key_buff2` follow a Gaussian distribution, we can design a distribution of the threads that provides both a good load balancing and locality of data. By default, OpenMP threads are scheduled dynamically to avoid unbalanced distribution of work, but the developers also propose a cyclic distribution of the threads over the loop. For our distribution, we split the loop into two equal parts and distribute each part among the threads in a round-robin way. This modification can be done by simply changing one line of code, the `#pragma omp` before the parallel loop.

With this code modification, we obtain the access distribution shown in the bottom of Figure 6. We can see that now each thread accesses a different part of `key_buff1`. Furthermore, if most of the accesses still occur in the middle of the structure, the average number of access across the structure is the same for all threads, which means that our distribution preserves the good load balancing. Our modification has also changed `key_buff2`'s accesses distribution. We can see that each thread uses mostly one part of the array and again the load balance is preserved.

The main point of our code modification is to improve the affinity between thread and memory, therefore we need to pin each thread on a core to keep them close to the data they access. *TABARNAC* also shows us that the first touch is always done by the thread actually using the data for IS, therefore we do not need to explicitly map the data to the NUMA nodes.

We compare the execution time of *IS* (class *D*) for the

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

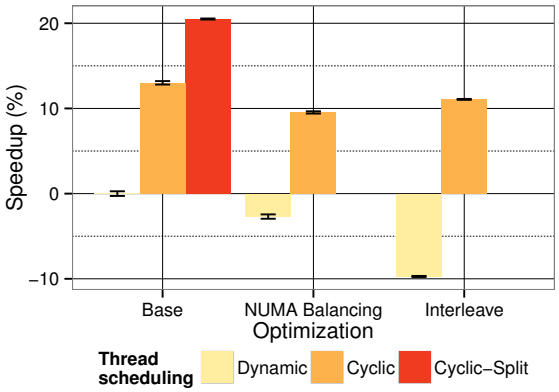


Figure 7: Speedup for *IS* (class D) compared to the baseline.

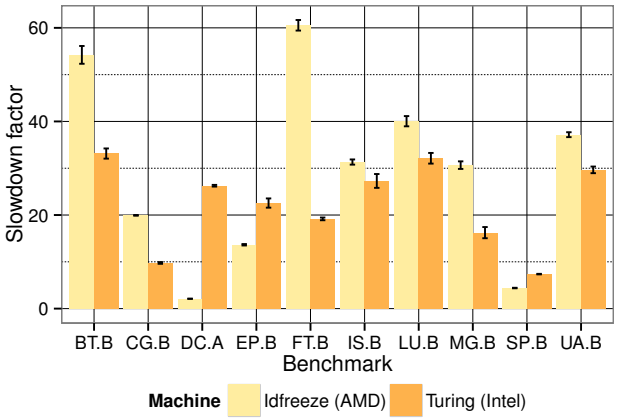


Figure 8: *TABARNAC*'s instrumentation overhead.

three scheduling methods, *Dynamic*, *Cyclic* with a step of 1 and *Cyclic-Split*: cyclic with the proposed distribution. For the two first methods, we compare the execution time on the base operating system, the interleave policy and with NUMA balancing enabled. As we map threads manually, interleave and NUMA Balancing are not relevant with our modifications and are therefore not evaluated.

Figure 7 shows the speedup of *IS* compared to the default version (*Dynamic*) for each scheduling method and for each optimization technique. The first thing to notice is that with the default *Dynamic* scheduling, both Interleave and NUMA Balancing slow the application down, by up to 10%. This shows that simple optimization policies can actually reduce performance for NUMA-unaware code. The *Cyclic* scheduling, proposed in the original code, already provides up to 13% of speedup. We can see that both interleave and NUMA Balancing are not suitable for this scheduling, since they reduce the performance gains. The *Cyclic-Split* version provides more than 20% of speedup with a very small code modification. This example shows how analyzing an application's memory behavior can lead to significant execution time improvement on an already optimized application where automatic techniques can actually slow the application down.

5.3 Overhead Analysis

Our last experiment aims at evaluating the instrumentation cost of *TABARNAC*. To do so, we executed all of the NAS Parallel Benchmarks in class *B* with 64 threads on both evaluation systems and compared the original execution time to the execution time with instrumentation enabled.

As we can see in Figure 8, on the Intel machine, the instrumentation slows the execution down by a factor from 10 to 30. On the AMD machine, the overhead is almost always higher, and for pathological cases, is two to three times slower than on the Intel machine. Although this overhead is not negligible, we have to consider the fact that often we can instrument smaller versions of the applications, as we focus on the general behavior. Moreover our method is more precise than sampling and thus one run is often enough. Finally, as our analysis is designed to be used during the development phase and at runtime in an automated tool, we consider that this overhead is acceptable.

5.4 Summary of Results

Our experiments have highlighted the fact that although automated tools such as NUMA Balancing can be efficient, in some cases they result in performance losses. Moreover, although simple static mapping policies can result in substantial improvements, the best policy depends on the memory accesses behavior of the parallel application. Therefore, it is necessary to understand this behavior to select the most appropriate mapping policy.

Our tools and methodology enables developers and users to achieve performance improvements in two ways. First, by providing a deep understanding of the memory access behavior, it enables the user to find the best mapping policy. Second, this knowledge can be used to identify and fix inefficient memory behavior. Our experiments showed that both situations result in significant performance gains.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented *TABARNAC*, a set of tools to analyze and optimize the memory behavior of parallel applications running on NUMA machines. We provide a custom memory tracer based on the Pin dynamic binary instrumentation tool which records the number of memory reads and writes performed by all threads for each data structure. The advantage of instrumentation is that it is the most accurate and portable way to generate memory traces. Despite the overhead caused by the instrumentation, our tool is efficient enough to analyze even huge applications in a reasonable time.

While other tools show how many remote access are triggered by which NUMA node, line of code or data structure, we provide information on how data structures are accessed. This information allows the user to understand *why* performance issues occur. *TABARNAC* presents this information through several meaningful yet readable plots. Each plot is preceded by explanations on how to read it, what kind of memory access issues it can help to identify and how to solve them.

We analyzed two parallel applications with *TABARNAC*: *Ondes3D*, a real life application that simulates seismic waves, and *IS* from the NAS Parallel Benchmarks which is known for being memory intensive with a random memory access

pattern. For both applications, *TABARNAC* helped us understand their performance issues. Using this knowledge, we proposed simple code modifications to optimize the memory behavior resulting, for each application, in significant speedups compared to the original version (up to 60% speedup) Improvements were also substantially higher than those provided by automated tools.

Future work will move in two directions. First, we will improve the structure detection support to be able to analyze Fortran programs, as many scientific applications are written in Fortran. Second, we will improve the detection of inefficient memory access behavior, such as an all-to-all sharing, to make the analysis partly automatic.

7. REFERENCES

[1] 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*, 22(6):685–701, 2010.

[2] AMD. AMD Opteron™ 6300 Series processor Quick Reference Guide. Technical Report August, 2012.

[3] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 319–330, 2010.

[4] 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*, SIGMETRICS '08, pages 229–240. ACM, 2008.

[5] R. Bosch, C. Stolte, D. Tang, J. Gerth, M. Rosenblum, and P. Hanrahan. Rivet: A Flexible Environment for Computer Systems Visualization. *SIGGRAPH Comput. Graph.*, 34(1):68–73, feb 2000.

[6] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180–186, Feb 2010.

[7] J. Corbet. Toward better NUMA scheduling, 2012.

[8] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quéma, and M. Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 381–393, 2013.

[9] L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia. SIGMA: a simulator infrastructure to guide memory analysis. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 1–13, Nov 2002.

[10] L. A. DeRose. The Hardware Performance Monitor Toolkit. In *Euro-Par 2001 Parallel Processing*, volume 2150, chapter Lecture Notes in Computer Science, pages 122–132. Springer Berlin Heidelberg, 2001.

[11] M. Diener, E. H. M. Cruz, and P. O. A. Navaux. Locality vs . Balance: Exploring Data Mapping Policies on NUMA Systems. In *International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 9–16, 2015.

[12] M. Diener, E. H. M. Cruz, P. O. A. Navaux, A. Busse, and H.-U. Heiß. kMAF: Automatic Kernel-Level Management of Thread and Data Affinity. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 277–288, 2014.

[13] M. Diener, E. H. M. Cruz, L. L. Pilla, F. Dupros, and P. O. A. Navaux. Characterizing Communication and Page Usage of Parallel Applications for Thread and Data Mapping. *Performance Evaluation*, 88-89(June):18–36, 2015.

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

[15] P. J. Drongowski. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. Technical report, AMD CodeAnalyst Project, 2007.

[16] P. J. Drongowski. An introduction to analysis and optimization with AMD CodeAnalyst™ Performance Analyzer. Technical report, 2008.

[17] F. Dupros, H. Aochi, A. Ducellier, D. Komatitsch, and J. Roman. Exploiting Intensive Multithreading for the Efficient Simulation of 3D Seismic Wave Propagation. In *IEEE International Conference on Computational Science and Engineering (CSE)*, pages 253–260, 2008.

[18] 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*, SC '14, pages 166–176. IEEE Press, 2014.

[19] Intel. Intel Performance Counter Monitor - A better way to measure CPU utilization, 2012.

[20] 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*, pages 22–30, Oct 2014.

[21] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS Parallel Benchmarks and Its Performance. Technical Report October, NASA, 1999.

[22] A. Kleen. An NUMA API for Linux, 2004.

[23] R. Lachaize, B. Lepers, and V. Quema. MemProf: A Memory Profiler for NUMA Multicore Systems. In *USENIX 2012 Annual Technical Conference (USENIX ATC 12)*, pages 53–64. USENIX, 2012.

[24] D. Levinthal. Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors. Technical report, 2009.

[25] 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*, PPoPP '14, pages 259–272. ACM, 2014.

[26] H. Löf and S. Holmgren. affinity-on-next-touch:

Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In *International Conference on Supercomputing (SC)*, pages 387–392, 2005.

[27] 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*, PLDI '05, pages 190–200. ACM, 2005.

[28] 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*, pages 11–22, Sept 2013.

[29] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. L. Scott. Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems. In *International Parallel Processing Symposium (IPPS)*, pages 480–485, 1995.

[30] 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*, SIGMETRICS '92/PERFORMANCE '92, pages 1–12. ACM, 1992.

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

[32] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, F. M. Quintão Pereira, and F. Magno. Compiler support for selective page migration in NUMA architectures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 369–380, 2014.

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

[34] C. P. Ribeiro, J.-F. Méhaut, A. Carissimi, M. Castro, and L. G. Fernandes. Memory Affinity for Hierarchical Shared Memory Multiprocessors. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 59–66, 2009.

[35] J. Tao, W. Karl, and M. Schulz. Visualizing the Memory Access Behavior of Shared Memory Applications on NUMA Architectures. In V. Alexandrov, J. Dongarra, B. Juliano, R. Renner, and C. J. K. Tan, editors, *Computational Science - ICCS 2001*, volume 2074, chapter Lecture Notes in Computer Science, pages 861–870. Springer Berlin Heidelberg, 2001.

[36] 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*, pages 42–49, Nov 2014.