NumaMMA: NUMA MeMory Analyzer

François Trahay, Manuel Selva, Lionel Morel, Kevin Marquet

NumaMMA: NUMA MeMory Analyzer

François Trahay
SAMOVAR, CNRS, Télécom SudParis, Université Paris-Saclay
Evry, France
francois.trahay@telecom-sudparis.eu

Manuel Selva
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
Grenoble, France
manuel.selva@inria.fr

Lionel Morel
Univ Grenoble Alpes, CEA, List
Grenoble, France
lionel.morel@cea.fr

Kevin Marquet
Univ Lyon,INSA Lyon, Inria, CITI
Villeurbanne, France
kevin.marquet@insa-lyon.fr

ABSTRACT

Non Uniform Memory Access (NUMA) architectures are nowadays common for running High-Performance Computing (HPC) applications. In such architectures, several distinct physical memories are assembled to create a single shared memory. Nevertheless, because there are several physical memories, access times to these memories are not uniform depending on the location of the core performing the memory request and on the location of the target memory. Hence, threads and data placement are crucial to efficiently exploit such architectures. To help in taking decision about this placement, profiling tools are needed. In this work, we propose NUMA MeMory Analyzer (NumaMMA), a new profiling tool for understanding the memory access patterns of HPC applications. NumaMMA combines efficient collection of memory traces using hardware mechanisms with original visualization means allowing to see how memory access patterns evolve over time. The information reported by NumaMMA allows to understand the nature of these access patterns inside each object allocated by the application. We show how NumaMMA can help understanding the memory patterns of several HPC applications in order to optimize them and get speedups up to 28% over the standard non optimized version.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Software and its engineering** → **Software performance**;

KEYWORDS

Performance analysis, NUMA architectures, Data and threads placement, Memory sampling

1 INTRODUCTION

Because symmetric memory architectures do not scale up with the number of cores, modern multicore architectures have non-uniform memory access (NUMA) properties: a given core accesses some memory banks faster than other banks. Typically, one finds two levels of addressable memory: one core may access a local memory really fast, or access a remote memory via an interconnect at some extra cost in time. A set of cores together with its local memory is named a NUMA *node*.

From the programs' perspective, this type of hardware requires to decide the *locality* of data and threads. Clearly, as many memory accesses as possible should be local to decrease there latency. But applications may comprise a large number of threads and many data dependencies between these threads. This is particularly true for High-Performance Computing (HPC) applications. It is then often not possible to avoid remote memory accesses when threads are sharing data as it would be highly inefficient to execute all threads on the same core, or even on cores on the same NUMA node.

As the number of threads and the amount of shared data increase, deciding the placement of data and threads also requires to take into account contentions that may appear on shared resources. When such contentions occur, it has been shown that interleaving memory pages on all the NUMA nodes is an efficient solution [16].

Recent results also show that NUMA architectures are asymmetric [21]: accesses from a core on a node A to a bank on node B may have performances completely different from accesses from a core on node B to a bank on node A. Moreover, the documentation of processors actually lacks details about this kind of characteristics and the programmer is thus left with trying out different combinations and infer system-level performance characteristics for application.

NUMA architectures are very complex indeed and they differ very much from one another. They have different numbers of nodes, different memory latencies, bandwidths and sizes, different symmetry models. As programmers are not likely to know those architectural details, we cannot expect them to port applications from one architecture to another and take the full benefits of the new architecture in terms of performance. On the other hand, hardware platforms are generic and cannot be expected to take specific applications needs into account. Hence, tools should be provided to help the programmer decide the locality of data and threads.

In the context of HPC, most applications have similar memory access patterns from one execution to another. Thus, by analyzing the memory access pattern of an application, a developer can figure out how data and threads should be placed on a given NUMA system in order to maximize local memory access and to reduce the memory contention on the NUMA

nodes. Determining the memory access pattern of an application can be done by analyzing its source code. However, this task may be tedious for large applications, and it is hard to grasp the impact of each memory object on the performance: some objects are rarely accessed and improving their locality is worthless, while some objects are accessed often and their placement have a significant impact on the overall performance of an application. Thus, memory profiling tools are needed to help understanding these pattern, and to guide the decision of locality of data and threads.

In this paper, we propose NUMA MeMory Analyzer (NumaMMA), a tool allowing to understand in details the memory access patterns of an application. NumaMMA first uses hardware capabilities to gather a memory trace. This trace is then processed offline in order to assess the cost of memory accesses to every object of the application and to know how threads access different parts of these objects. Understanding memory access patterns inside objects is crucial in the context of HPC applications because they often only access to few very large objects a huge number of time. We then make the following contributions:

- we propose an open-source tool[1] able to report how memory access patterns inside objects evolve over time;
- we combine this reporting with an efficient trace collection mechanism based on hardware sampling;
- we provide developers with original visualization means of these memory access patterns;
- we show that this information can be used for defining a placement strategy that improves the performance of applications by up to 28%.

The remainder of the paper is organized as follows. Section 2 details related work. Section 3 then explains the contributions we make. Section 4 evaluates experimentally NumaMMA. Section 5 concludes and gives perspectives on this work.

2 RELATED WORK

With the increasing number and complexity of NUMA machines, the placement of thread and data in these machines has gained strong focus in the last decades. When targeting NUMA machines, threads and data placement must first limit the number of remote memory accesses. Second, it must ensure there is no contention on shared resources such as the interconnect network allowing any core to access any memory bank. Threads and data placement is generally handled either explicitly by the programmer at the application level or automatically at runtime by the operating system potentially with help from the compiler. To help the programmer understand the memory access patterns of applications and then optimize them, many profiling tools have been proposed.

We now quickly review runtime approaches before presenting in details existing NUMA profiling tools. For a complete survey of solutions that have been proposed for the threads and data placement problem, the reader should refer to the recent work of Diener et al. [14].

[1]https://github.com/numamma/numamma

2.1 Runtime Placement

Runtime mechanisms for placing threads and data are required in contexts where applications are not known. In this case, the operating system [3, 9, 11, 12, 15, 18], or the runtime system between the operating system and applications [7, 8], is responsible for deciding where to allocate data in memory and on which core to execute which thread.

General purpose operating systems provides two main global policies regarding memory allocation. By default, Linux allocates all the memory pages of an application in the NUMA node where the first thread accessing it is running. This is the *first-touch* policy. Linux also offers the *interleaved* policy, where all the pages of an application are allocated in a round-robin fashion among all NUMA nodes. While being effective in some situations, these two global policies are not efficient at all for applications having complex memory access patterns. In particular, the memory access patterns of an application may not be the same for different objects and thus a global policy cannot be efficient. Also, these memory access patterns may evolve over time. As a result, the placement of a memory page may become misfit. The Solaris operating system offers the *next-touch* policy [3] that takes a memory page already allocated and migrates it in order to improve its locality with the thread that uses it. Integration of this next-touch policy into the Linux kernel has also been proposed [18, 20]. Starting from version 3.8 released in 2013, the Linux kernel also provides NUMA memory balancing [1]. This feature relies on periodically unmapping pages and later trapping a page fault to detect which thread access which memory.

Solutions have been proposed to improve the basic policies provided by standard operating systems. Several of the proposed solutions rely on hardware performance counters [8, 11, 15], or on custom hardware extensions [9] that are used to sample memory accesses. This sampling provides an insight about memory access patterns of the application. Based on this profiling information, pages and threads are then moved to increase memory locality and to avoid contention on the interconnect. Other solutions rely on the compiler providing information that can be used to infer memory access patterns before deciding where the data should be allocated [12]. Finally, some solutions rely on the knowledge of the affinity between OpenMP threads to improve the locality of threads and data: the threads that belong to the same OpenMP team are likely to access the same memory regions, and grouping them improves the locality [7].

All these solutions rely on runtime heuristics. The quantity of information that can be processed is thus limited so as to limit the overhead on the application's performance. This is a major difference with our approach where profiling is performed offline: we can build much more precise performance metrics that can be used to take better placement decisions.

2.2 Offline Placement

Over the years, many works have focused on collecting memory traces in order to analyze them offline. These solutions differ in the way they collect memory access information, in

the way they process this information and in the way they use it ultimately.

2.2.1 Collecting Memory Traces. Multiple solutions have been proposed for collecting memory traces. Memory accesses can be captured by simulating the execution of an application [10]. However, the simulating time is prohibitive, which prevents from using this approach on most applications.

Another solution consists in instrumenting the application binary with tools like Pin [24] so that each instruction that reads or writes data is recorded [5, 13, 28, 30]. While the overhead caused by the instrumentation is reduced compared to the simulation approach, it still causes the application to run up to 20 times slower than the non-instrumented version. This prevents from using this solution on large applications.

A third approach is to leverage the memory sampling capabilities provided by the hardware. Modern processors implement hardware-based monitoring systems (such as Intel PEBS, or AMD IBS) that periodically save information about the instruction being executed. This mechanism can be used for collecting the memory addresses that are accessed [17, 19, 22, 23, 25–27, 31]. Since only some of the instructions are collected, this approach is less precise than instrumentation but more efficient. Also, the collection of a single sample is more efficient than instrumentation because it is done by the hardware. Collecting a sample with Intel PEBS only requires 200 to 300 nanoseconds [2]. The impact on the application performance is thus small. This approach is thus applicable on large applications. Also, compared to the approach based on binary instrumentation, hardware-based sampling allows to record the the level in the memory hierarchy (L1, L2, ...) that served an access along with the latency of the access.

2.2.2 Existing Tools for Thread and Data Placement. Molina da Cruz et al. propose an automatic placement tool [10]. This tool relies on simulation to trace all memory accesses. It then builds a complete graph representing the amount of communication between threads. A partitioning algorithm is then applied on the graph to know which threads should be put close together.

Song et al. proposed an automatic solution, for thread placement only, using binary instrumentation to collect traces [30]. Their solution also builds a graph representing the amount of communication between threads. It then applies a partitioning algorithm on this graph to decide where threads should be placed. Numalize [13] also relies on binary instrumentation to perform automatic threads and data placement. For threads placement, Numalize uses a graph representing the amount of communication between threads. Partitioning is done using that graph. For data placement, Numalize computes high level metrics at application level. Compared to other solutions, Numalize is the only one taking the time dimension into account. Said differently, Numalize computes threads and data placement for different execution phases of the application. Tabarnac [5] is another tool relying on binary instrumentation. Tabarnac offers visual representations allowing to inspect how threads access internal pages of large objects. These representations are global to the entire execution of the application, they do not include time information. RTHMS [28] uses binary instrumentation for collecting memory access patterns on objects on a machine whose memory is heterogeneous such as the Intel Knight Landing architecture. In such machines, objects are either allocated on the fast but small memory, or on the slow but large memory. RTHMS then analyzes the collected data in order to decide where to allocate the objects, based on their expected impact on performance computed from the number of accesses, the life span and the read/write frequency.

Because simulation and binary instrumentation have a huge impact on performance, many tools have been proposed that rely on hardware sampling mechanisms [17, 19, 22, 23, 25–27, 31]. The work by Marathe et al. [25, 26], targeting Intel Itanium architectures, proposes an automatic page placement that allocates pages on the NUMA node where they are most used or on the node that will minimize the total cost of all its accesses. Memphis [27] and MemProf [19], both targeting AMD processors, provide NUMA related information to the programmer in a data-centric view, that is using objects of the application. For each object in the application, they report the total number and costs of remote memory accesses. No information about the distribution of accesses inside an object is reported. MemAxes [17], one of the most advanced visualization tool for pinpointing NUMA effects, focuses on providing information related to the application along with standard metrics on memory accesses. This is done with the help of the programmer who must instrument her application to give information to the tool. Also, similarly to Memphis and MemProf, MemAxes does not provide any visual information about the dsitribution of accesses inside large objects. Finally, HPCToolkit [22, 23] also reports information about NUMA effects over the application. Its main contributions are global metrics allowing to identify whether or not changing the data placement could lead to performance benefits along with a data-centric reporting of memory accesses. HPCToolkit also reports some information about memory accesses inside large objects. Nevertheless, this information only contains the minimum and maximum addresses inside the object accessed by each thread.

3 NUMAMMA

In this section, we present NumaMMA, a memory profiler that captures the memory access pattern of threads on objects in memory. The approach consists in executing an application offline, retrieving useful information about memory accesses, and providing this information to the application developer. She can then use this information to optimize the placement of threads and data.

During the profiled execution of the application, NumaMMA collects information on the dynamic allocations as well as on global variables. NumaMMA uses the sampling features provided by the hardware to collect samples of the memory accesses performed by the application threads. Relying on hardware memory sampling allows to capture the application behavior without prohibitive overhead.

After the execution of the application, NumaMMA searches for the memory object accessed by each collected sample and computes statistics on each object. Once all the samples are processed, NumaMMA reports statistics on the memory objects most accessed by the application. These statistics can be used by the application developer in order to tune the placement of the threads and data.

In this section, we first describe how memory access samples are gathered. Second, we describe how information on memory objects is collected. Then, we describe how samples are matched with memory objects and which statistics are computed. Finally, we describe how NumaMMA reports information about memory accesses to the user.

3.1 Collecting Samples

Modern processors implement sampling mechanisms able to collect information on executed instructions with a low overhead. Intel provides Precise Event-Based Sampling (PEBS) while AMD provides Instruction Based Sampling (IBS). When sampling, the CPU periodically records information on the instruction that is being executed in a dedicated memory location and notifies the software through an interrupt. With PEBS, it is possible to configure the hardware to only sample memory reads and/or memory writes. To limit the overhead of the sampling, it is also possible with PEBS to collect several samples before notifying the software.

During the profiled execution of the application, NumaMMA uses `numap` [29] for collecting memory samples. Note that `numap` currently only supports Intel PEBS, but it could be extended to other sampling mechanisms such as AMD IBS. Also note that Intel micro architectures prior to Sandy Bridge (2011) only support the sampling of read accesses. The precision of the information gathered through sampling will thus depend on the profiling capabilities offered by the hardware platform used. Each memory sample is composed of:

- the type of the access, that is read or write;
- the identifier of the thread that executes the access;
- the time at which the sample has been taken;
- the address accessed by the instruction;
- which part of the memory hierarchy was accessed, that is L1/L2/L3 cache or local memory or memory located on a remote NUMA node;
- the cost of the memory access, that is the latency.

`numap` records samples in a dedicated memory buffer. When this buffer is full, the recording of samples is stopped. Thus, NumaMMA needs to collect and flush the sample buffer regularly so as not to lose samples. To that end, when the application calls an allocation function (such as `malloc` or `free`), NumaMMA stops recording samples, copies the collected samples to an other location that will be analyzed *post-mortem*, and resumes recording samples. However, samples may still be lost if the application does not call allocation functions regularly. To mitigate the loss of samples, NumaMMA can also set an alarm to periodically collect samples. The usage and impact of this parameter is discussed in Section 4.

3.2 Identifying Memory Objects

In order to find the memory object accessed by a sample, NumaMMA collects information on dynamic allocations as well as on static objects, that is global variables. For each memory object, NumaMMA stores the base address of the object and its size, the allocation timestamp, and the de-allocation timestamp. Memory objects are stored in a binary tree ordered by the object address.

Dynamic allocations are tracked by overloading the main memory allocation functions: `malloc()`,`realloc()`,`calloc` and `free()`. When the application allocates an object, NumaMMA intercepts the function call using the `LD_PRELOAD` mechanism, and stores information on the allocated objects. For dynamic objects, NumaMMA also collects information on the allocation call sites, that is the function and the line at which the object was allocated.

NumaMMA collects information on static objects at the application start up. It reads the list of symbols in the ELF file, and searches for global variables and the corresponding size and offset. NumaMMA then determines at which address the ELF file is loaded in order to compute the address of the variable in the current address space. This computing step is required for code compiled in a position independent way.

3.3 Processing Samples

After the execution of the application, the collected samples are processed in order to identify the memory object corresponding to each sample. To do so, NumaMMA browses the binary tree that contains the memory objects and searches for an object whose address range includes the address reported in the sample.

Because of the dynamic allocation, several memory objects may match an address. This happens when the application allocates with `malloc` an object o_1, free it, and allocates another object o_2. In this case, `malloc` may allocate o_2 at the same address *addr* as o_1. Thus, when searching for the object that corresponds to address *addr*, NumaMMA will identify both o_1 and o_2. Thus, NumaMMA also compares the timestamp of the sample with the allocation and free dates for o_1 and o_2. Also, this implies that nodes of the binary tree recording memory objects are lists of objects (all allocated at the same address) and not a single object.

Once the memory object that matches a sample is identified, NumaMMA updates the following counters associated to the object:

- the number of read/write accesses;
- the number of read/write accesses to/from a remote NUMA node;
- the total read/write accesses cost.

These counters are computed both globally and in a per-thread basis. Since multiple threads may access different portions of a single object, and we are particularly interested in understanding how, NumaMMA also computes these counters for each memory page, 4 KiB by default or 2 MiB when using huge pages.

3.4 Reporting Memory Access Information

After all the samples are processed, NumaMMA outputs several results. All the outputs of NumaMMA can be either global, that is including read and write accesses, or specific to read or to write accesses. First, NumaMMA prints the list of memory objects along with their accumulated counters. By default, the list is sorted by the total number of accesses, but this can be changed to sort by the total memory cost. Also, NumaMMA groups in this list the objects that have been allocated at the same call site. There counters are summed and they are reported as a single object. This choice is motivated by the fact that changing the allocation at the callsite line will change the placement of all the objects allocated at this callsite. This list of objects provides application developers with useful information on which objects are the most likely to affect the performance of the application.

For each object, NumaMMA textually reports the number of detected memory accesses per page and per thread. This information can also be reported in a graphical way using a communication matrix, as shown in Figure 1. This matrix shows which thread accesses to which memory page. The color indicates the number of accesses. In this example, thread #0 mainly accesses pages from 0 to 1000 and from 4200 to 5100, while thread #1 mostly access pages 1000 to 1800 and from 5100 to 6000. From this graph it is clear that threads only access specific sub-parts of the `main_flt_mem` object. Using this information, we may decide to place the threads that work on the same memory pages on the same NUMA node. Memory pages could also be bound to NUMA nodes according to the thread access pattern.

NumaMMA also textually reports, for each object, the list of samples that were matched with the object. This provides more detailed information on an object, including the time dimension. Figure 2 shows how NumaMMA can report this information graphically. This plot shows the memory accesses of the threads during the execution of the application. The horizontal axis represents the time and the vertical one represents the offset in the memory object. Each point corresponds to a single memory access sample, and the color represents the thread accessing the object. By adding a temporal dimension, this view completes the information provided by the one shown in Figure 1. We clearly see that the two different subparts of the `main_flt_mem` object accessed by each thread are accessed in different stages of the application. If for any reason, the upper part of the data is allocated on a different NUMA node that the lower part, then we can use the timing information to migrate the four threads on the node where the upper part is just before starting the second execution stage.

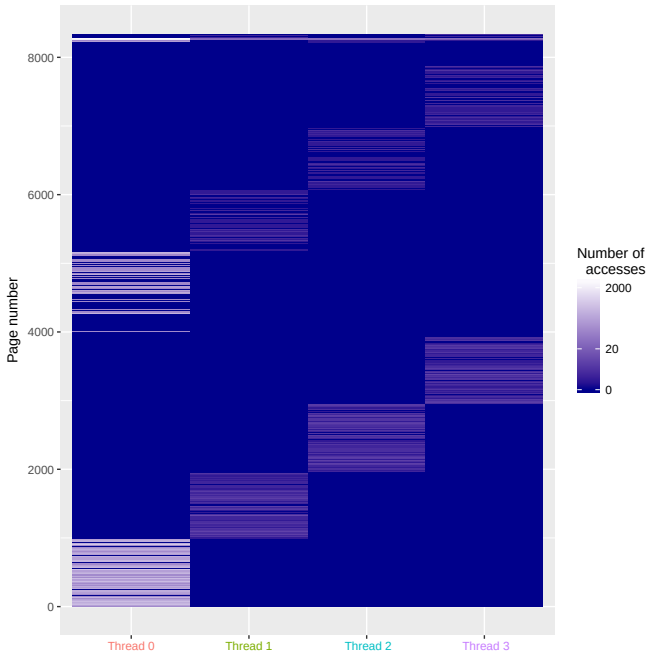

Figure 1: Number of, per thread, memory accesses to the pages of the `main_flt_mem` object in NPB CG, class A, 4 threads. Threads access different sub-parts of the object.

4 EVALUATION

In this section, we evaluate and show how NumaMMA can help understanding memory access patterns to improve performance. We demonstrate this on applications from the NAS Parallel Benchmarks and on Streamcluster from PARSEC. We used two NUMA machines:

- Intel32 has 2 Intel Xeon E5-2630 v3 processors with 8 cores/16 threads in each (total: 16 cores/32 threads), running at 2.4GHz. The machine is equipped with 2 NUMA nodes connected through QPI and 32GB of RAM. It runs Linux 4.11, glibc version 2.24, and GCC 6.3;
- Amd48 has 4 AMD Opteron 6174 processors with 12 cores in each (total: 48 cores) running at 2.2GHz. The machine is equipped with 8 NUMA nodes (2 nodes per processor) connected through HyperTransport 3.0, and 128GB of RAM. It runs Linux 4.10, glibc version 2.25, and GCC 6.3.

The applications were compiled with the `-O3` flag and were not stripped so that NumaMMA can find the list of global variables in the application. The NAS Parallel Benchmarks use the GNU OpenMP shipped with GCC 6.3. Streamcluster was compiled with the `-g` flag so that NumaMMA can identify the file and line of each memory object using the debugging information.

4.1 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) [4] is a suite of HPC kernels. We run the OpenMP implementation of NPB 3.3, class C, on the Intel32 machine.

4.1.1 NumaMMA Overhead. We first run NPB kernels with and without NumaMMA in order to assess the overhead of NumaMMA. The NAS kernels are executed with 32 threads

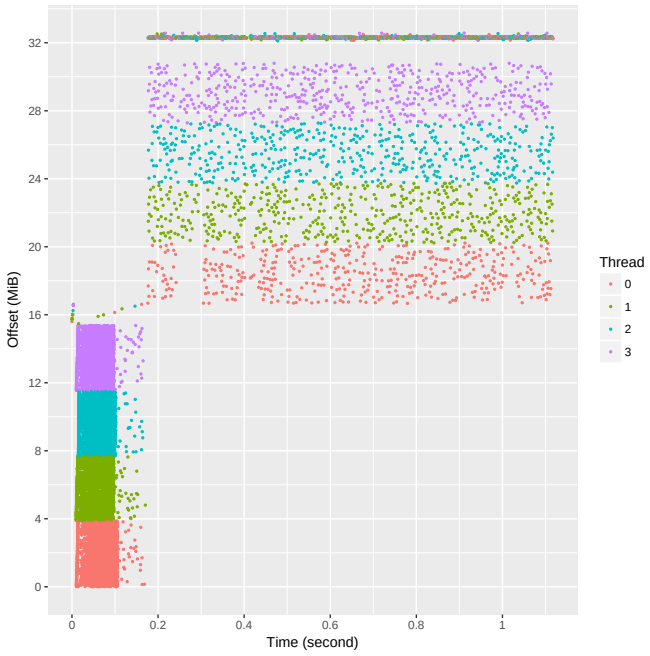

Figure 2: Evolution over time of, per thread, memory accesses to the pages of the `main_flt_mem` object in NPB CG, class A, 4 threads. Threads access different sub-parts of the object at different stages in time.

and we use `hwloc` [6] and `GOMP_CPU_AFFINITY` so that consecutive threads are located close to each other in the machine topology. We run the kernels with two different settings:

- *NumaMMA_2k*: sampling rate of 2000. We record a memory sample every 2000 memory accesses, and the alarm system is disabled, that is samples are collected when the application calls an allocation function;
- *NumaMMA_10k*: sampling rate of 10000, and alarm period of 100 milliseconds, that is samples are collected when the application calls an allocation function and every 100 milliseconds. The collection on the alarm is done in every cases, independently of the number of calls to allocation functions.

To evaluate the overhead of NumaMMA on the application run time, we compare the execution time of NAS Parallel Benchmark kernels when run with and without NumaMMA. The results of this experiment are reported in Table 1. The results show that NumaMMA has little effect on three kernels (LU, IS, SP). Two kernels (UA and MG) are more affected by NumaMMA. Nevertheless the overhead is less than 12% in all cases. This remains low compared to other techniques that rely on software mechanisms such as dynamic binary translation to track memory accesses [5, 13, 30] or simulation [10]. This maximum overhead of 12% is not prohibitive as it only affect the debugging phase of the application development.

The overhead has three main causes:

- the overhead of sampling instructions which is influenced by the sampling rate;

kernel	*native* time(s)	*NumaMMA_2k* time(s)	*NumaMMA_2k* ovhd(%)	*NumaMMA_10k* time(s)	*NumaMMA_10k* ovhd(%)
BT.C	81.3	82.3	1.30	85.7	5.46
CG.C	21.6	23.3	8.01	22.2	2.93
EP.C	10	10.5	3.81	10.5	4.40
FT.C	19.6	20.2	3.14	21.7	10.77
IS.C	1.5	1.48	-2.11	1.45	-4.35
LU.C	61.8	58	-6.11	61.7	-0.12
MG.C	10.4	11.3	9.01	10.9	5.48
SP.C	168.6	169.8	0.68	169.6	0.59
UA.C	86.6	93.5	8.00	96.5	11.37

Table 1: Overhead of NumaMMA, depending on sampling frequency, on NPB kernels class C. The overhead is below 12% in all cases.

kernel	*NumaMMA_10k* nsamples (million)	*NumaMMA_2k* nsamples (million)	*NumaMMA_2k* nstack (million)	*NumaMMA_2k* pstack (percentage)
BT.C	172	0.7	0.4	54.55
CG.C	41	0.7	0.09	12.41
EP.C	19	0.6	0.59	99.76
FT.C	39	0.5	0.22	43.88
IS.C	3	0.18	0.17	96.01
LU.C	110	0.65	0.23	35.19
MG.C	22	0.8	0.28	35.68
SP.C	334	0.5	0.09	17.29
UA.C	171	0.6	0.19	30.92

Table 2: Number of samples collected on NPB. *NumaMMA_2k* gives a "high definition" partial view of the application memory access pattern while *NumaMMA_10k* gives a "low definition" complete view of the application memory access patterns.

- the cost of copying samples which is influenced by the number of collected samples and thus by the sampling rate;
- the overhead of intercepting dynamic allocation calls which is influenced by the number of memory allocations.

The two first causes of overhead are closely related. More samples will be copied if the sampling rate is high. Also, the overhead resulting from these two causes is proportional to the number of memory access instructions performed by the application compared to the total number of instructions. The overhead of intercepting dynamic allocation is not significant in the NPB kernels because there are very few such allocations.

4.1.2 NumaMMA Accuracy. Table 2 reports the number of samples that were collected by NumaMMA, as well as the number of samples that correspond to an address on the stack when run with *NumaMMA_2k*. The results show that *NumaMMA_10k* collects millions of samples, while *NumaMMA_2k* captures up to half a million samples. This difference is caused by their respective configuration. In *NumaMMA_2k*, the CPUs

capture samples frequently, and the sample buffer quickly becomes full. When this happens, the sampling of memory accesses is stopped. Thus, *NumaMMA_2k* gives a "high definition" partial view of the application memory access patterns. We could have used a sampling rate of 2000 along with an alarm to have a complete high definition view of the application at the price of an higher overhead. Nevertheless, as shown in the next section, we are able to understand LU patterns and then optimize it using *NumaMMA_2k*. *NumaMMA_10k* is configured to capture samples less often, but the alarm every 100 ms empties the sample buffer frequently to reduce the number of lost samples. This gives a "low definition" complete view of the application memory access patterns.

The results reported in Table 2 also show that a part of the collected samples correspond to memory addresses on the stack of the application threads. For these stack accesses, nothing should be done because the default operating system first touch policy ensures local accesses. Kernels making mostly stack accesses, EP and IS, could not benefit from memory optimization while all the others may be optimized.

4.1.3 Analysis And Optimization of LU. Based on the data collected with *NumaMMA_2k*, NumaMMA identifies that LU threads access mainly three objects:

- `cvar` which size is 558 MiB and which represents 58%of the samples;
- `cexact` which size is 520 bytes and which represents 23% of the samples;
- `cjac` which size is 20 MiB and which represents 10% of the samples.

`cexact` is a small object with a size smaller than one pages for which NumaMMA reports accesses that are randomly distributed across all the threads. We conclude that little can be done to improve the allocation strategy for this object.

Regarding `cvar` and `cjac`, these are composed of many memory pages. We now use NumaMMA graphical representations to observe and try to understand how threads access these two objects. The access patterns of `cvar` and `cjac` are reported respectively in Figure 3 and Figure 4. These figures report the access patterns over a short period of time compared to the total execution time of the kernel. Figure 3 represents 0.88% of the total execution time (58 seconds) while Figure 4 represents 0.1%. The reported patterns are repeated over all the iterations of the application and the figures are thus sufficient to understand the memory behavior of the whole. In Figure 3 we clearly see patterns evolving over time while the access behavior is constant in Figure 4. The main characteristic of all the patterns for both objects is the presence of colored horizontal lines. We also notice that the color order of these lines corresponds to the order of thread identifiers and that it is repeated several time. This means that threads are accessing only sub-parts of the `cvar` and `cjac` objects in a cyclic fashion. Because the height of the observed cycles are different for the two objects, it suggests that the optimal memory placement policy is different for the two variables:

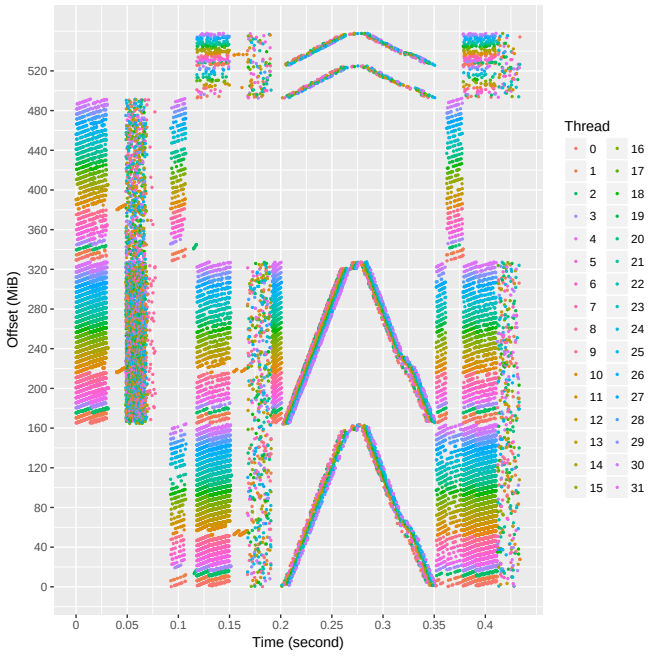

Figure 3: Evolution over time of, per thread, memory accesses to the pages of the `cvar` object in NPB LU, class C, 32 threads. The pattern shown here over half a second is repeated over all the iterations of the kernel.

- for `cvar` the placement must take into consideration the fact that the memory accesses are uniformly distributed to all threads on blocks of 160 MiB;
- for `cvar` the placement must take into consideration the fact that the memory accesses are uniformly distributed to all threads on blocks of 5 MiB.

To assess the effect of memory placement in LU, we run the application on AMD48 and we apply several binding policies for `cvar` and `cjac`. Since `cvar` and `cjac` are global variables, NUMA allocation functions cannot be used. Thus, we create a library that loads at the application startup and applies binding policies using `mbind`. In the following, a *block distribution* of size N MiB means that pages are allocated by blocks of N MiB. The pages of the first N MiB are allocated on the first NUMA node, the ones of the second N MiB on the second node, etc. When the last node has been reached, the block distribution starts again from the first node.

We implemented 4 allocation policies for the `cvar` and `cjac` objects:

- *first-touch*: the pages of the two objects are allocated with Linux default *first-touch* policy;
- *interleaved*: the pages of the two objects are allocated to all the NUMA nodes in an interleaved fashion;
- *block-naive*: the pages of the two objects are allocated to all the NUMA nodes using a block distribution which size is the object size divided by the number of NUMA nodes, that is 8 in our case. This policy naively relies on the assumption that the work will be distributed among

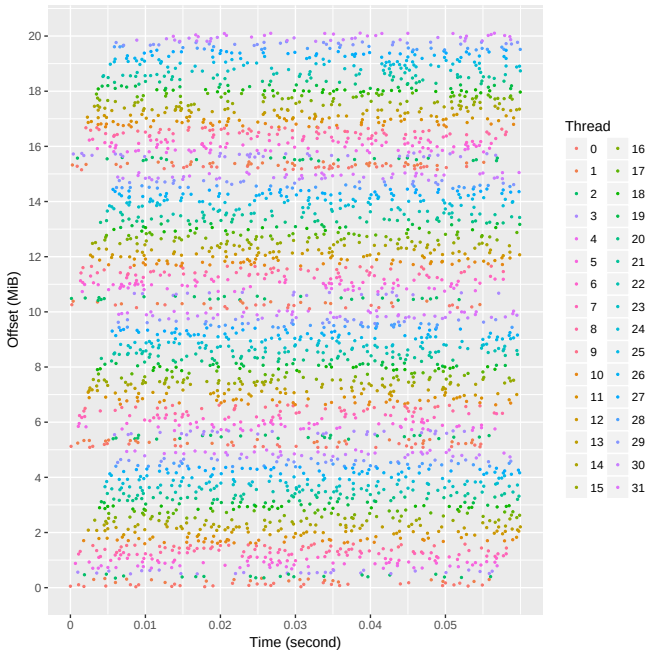

Figure 4: Evolution over time of, per thread, memory accesses to the pages of the `cjac` object in NPB LU, class C, 32 threads. The pattern shown here over a short period of time is repeated at the beginning of each iteration.

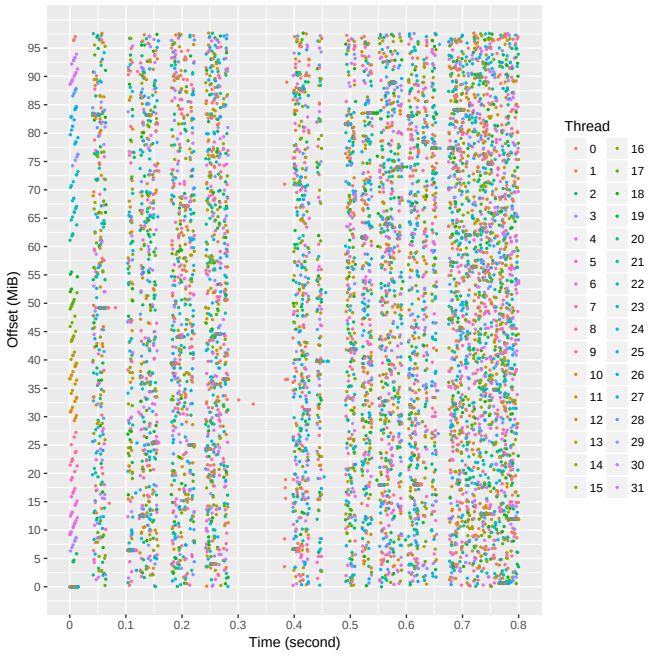

Figure 5: Evolution over time of, per thread, memory accesses to the pages of the `block` object in Streamcluster. The pattern shown here over 0.8 seconds is repeated over all the iterations of the kernel.

policy	execution time(s)	speedup
first-touch	102.53	1
interleaved	106.86	0.96
block-naive	109.88	0.93
NumaMMA	81.05	1.27

Table 3: Performance improvement on NPB LU. By allocating `cvar` and `cjac` in the way suggested by NumaMMA, we have a speedup of 27% over Linux default *first-touch* policy.

threads with the biggest possible blocks, such as in an OpenMP loop with static scheduling;

- *NumaMMA*: the pages of `cvar` and `cjac` are allocated according to the objects access patterns as identified previously. `cvar` pages are allocated with a 20 MiB block distribution such that the 160 MiB blocks are spread over the 8 NUMA nodes. For `cjac` we use a 2 MiB block distribution. Ideally, the 5 MiB blocks would be spread over the 8 NUMA nodes leading to a block size of 0.625 MiB, but due to the use of huge pages, the granularity for data placement is 2 MiB.

The pages of all other objects are allocated according to the *first-touch* policy used by default by the operating system.

The results we obtained on LU are reported in Table 3. The *NumaMMA* enabled policy results in a significant 27% gain while the classic interleaved as well as the naive block distribution slow the kernel down.

It is worth mentioning that while NumaMMA is not yet available for AMD architectures, we have been able to use it to optimize the LU kernel on the Amd48 machine by profiling it on the Intel32 machine. This has been made possible because of the way NPB kernels are implemented. In these OpenMP kernels, the loops that process the objects such as `cvar` and `cjac` are evenly distributed over the OpenMP threads. This means that the pattern observed for a particular number threads can be extrapolated to another number of threads.

4.2 Streamcluster

Streamcluster is a parallel application from the PARSEC benchmark suite. We run this application on the Intel32 machine with NumaMMA with a sample rate of 2000 and the alarm mechanism disabled. The total runtime of this profiled execution of the application is 125 seconds. NumaMMA collects 115.7 million samples, including 102.1 million (88%) on the stack and 13.6 millions (12%) on global and dynamically allocated objects and dynamically allocated ones. NumaMMA reports that two objects, both dynamically allocated with `malloc`, are mainly accessed:

- `block` which size is 98 MiB and representing 66% of the samples on global and dynamically allocated objects;
- `points` which size is 6 MiB and representing 31% of the samples on global and dynamically allocated objects.

Figures 5 and 6 depict the detected access patterns for `block` and `points` as reported by NumaMMA. The `block` variable is accessed randomly by all the threads. The access pattern for

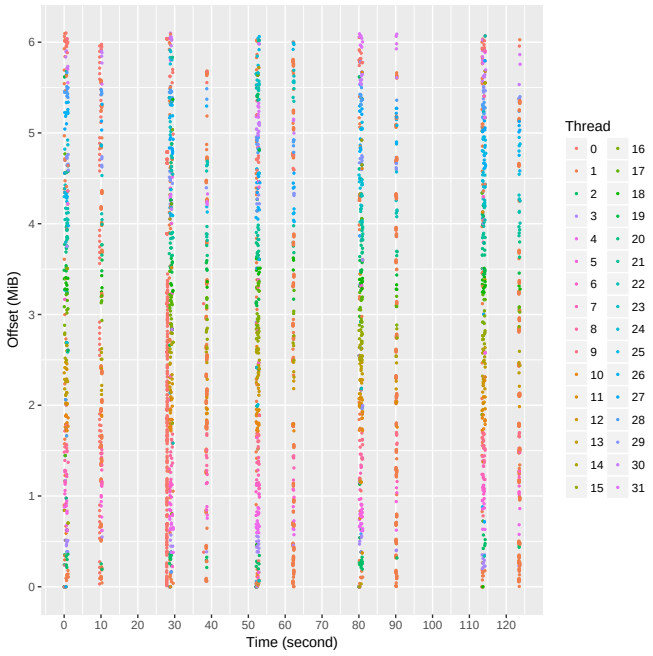

Figure 6: Evolution over time of, per thread, memory accesses to the pages of the `points` object in Streamcluster. The pattern shown here correspond to the complete execution of the application.

`points` is different where each thread processes a part of the object. These access patterns suggest that the optimal memory placement policy is different for the two variables:

- the pages `block` should be allocated using an *interleaved* policy so that memory access to this variable are spread over all the NUMA nodes to reduce the saturation of the interconnect;
- the pages `points` should be allocated using a *block-naive* distribution so that each thread access local memory.

To assess the impact of memory placement on this application, we run Streamcluster on the AMD48 machine with 48 threads using several placement policies. We implement 4 different policies for the `block` and `points` objects:

- *first-touch*: the pages of the two objects are allocated with Linux's default *first-touch* policy;
- *interleaved*: the pages of the two objects are allocated to all the NUMA nodes in an interleaved fashion;
- *block-naive*: the pages of the two objects are allocated on all the NUMA nodes using a block-naive distribution as in NPB lu in the previous section;
- *NumaMMA*: the pages of `block` are allocated using the *interleaved* policy, and the pages of `points` are allocated using a 768 KiB block distribution such that the 6 MiB of the object are spread over the 8 NUMA nodes.

The pages of all other objects are allocated according to the *first-touch* policy used by default by the operating system. We also pin the threads to cores so that consecutive threads are located close to each other in the machine topology.

policy	execution time(s)	speedup
first-touch	93.72	1
interleaved	76.76	1.22
block-naive	79.75	1.17
NumaMMA	73.32	1.28

Table 4: Performance improvement on Streamcluster. By allocating `block` and `points` in the way suggested by NumaMMA, we have a speedup of 28% over Linux default *first-touch* policy. This 28% speedup is better than the 22% one of the *interleaved* policy.

The results of this evaluation are reported in Table 4. While using a single policy (*block* or *interleaved*) for both objects improves the performance, the best performance is obtained when using the most appropriate policy for each object as inferred using NumaMMA. In this case we have a speedup of 28% which is better than the 22% one obtained with the interleaved policy. Again, for the same reasons than for LU described above, we have been able to optimize Streamcluster on the AMD48 machine by profiling it on the INTEL32 machine.

5 CONCLUSION AND FUTURE WORK

We have presented NumaMMA, a new memory profiler allowing to understand the evolution of memory access patterns inside objects allocated by applications. Compared to all existing offline profiling solutions presented in Section 2, NumaMMA is the first open-source software combining efficient trace collection using hardware sampling with the reporting of information at the page level to understand the distribution of memory accesses inside each object of the application. Also, NumaMMA provides original visualization means allowing to see how memory access patterns evolve over time.

The evaluation shows that the overhead caused by NumaMMA is low. The experiments also show that the memory access information collected by NumaMMA can be used for improving the allocation strategy of several applications from the NAS Parallel Benchmark and the Streamcluster benchmark. The optimization consists in allocating the application objects impacting performances, that is the most accessed ones, according to their access patterns as reported by NumaMMA. As a result, the optimized applications perform significantly better than the original ones.

We are already working on several extensions to NumaMMA. First, we want to provide an automatic way of computing the best memory allocation policy for any object. This includes computing automatically the size of the distribution of objects for which a block-distribution should be made. Second, we are planning to use the timing information provided by NumaMMA to implement runtime mechanisms allowing to dynamically adapt memory placement according to the application phases. Because moving pages at runtime is expensive, the number of such dynamic adaptations should be made as low as possible, and thus we must focus on long phases only.

Third, we are also working on the integration of the cost information of memory accesses, that is the latency, into the visual representations provided by NumaMMA. Finally, we started to work on the automatic implementation of the memory policies suggested by NumaMMA such that the programmer does not have to modify its application by hand.

ACKNOWLEDGEMENTS

This work was supported by the Paris Ile-de-France Region.

REFERENCES

[1] Last accessed 24th, May 2018. NUMA memory balancing in the Linux kernel. https://www.kernel.org/doc/Documentation/sysctl/kernel.txt. (Last accessed 24th, May 2018).

[2] Soramichi Akiyama and Takahiro Hirofuchi. 2017. Quantitative Evaluation of Intel PEBS Overhead for Online System-Noise Analysis. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*. 3.

[3] Joseph Antony, Pete P Janes, and Alistair P Rendell. 2006. Exploring thread and memory placement on NUMA architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In *Proceedings of the international conference on High Performance Computing, HiPC'06*. 338–352.

[4] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks. *The International Journal of Supercomputing Applications* 5, 3 (1991), 63–73.

[5] David Beniamine, Matthias Diener, Guillaume Huard, and Philippe O. A. Navaux. 2015. TABARNAC: Visualizing and Resolving Memory Access Issues on NUMA Architectures. In *Proceedings of the 2Nd Workshop on Visual Performance Analysis (VPA '15)*. ACM, New York, NY, USA, Article 1, 9 pages. https://doi.org/10.1145/2835238.2835239

[6] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010. hwloc: A generic framework for managing hardware affinities in HPC applications. In *Proceedings of the International Conference on Parallel, Distributed, and Network-Based Processing, PDP'10*.

[7] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. 2010. ForestGOMP: an efficient OpenMP environment for NUMA architectures. *International Journal of Parallel Programming* 38, 5 (2010), 418–439.

[8] Quan Chen and Minyi Guo. 2015. Locality-Aware Work Stealing Based on Online Profiling and Auto-Tuning for Multisocket Multicore Architectures. *ACM Trans. Archit. Code Optim.* 12, 2 (July 2015).

[9] Eduardo H. M. Cruz, Matthias Diener, Laércio L. Pilla, and Philippe O. A. Navaux. 2016. Hardware-Assisted Thread and Data Mapping in Hierarchical Multicore Architectures. *ACM Trans. Archit. Code Optim.* 13, 3 (Sept. 2016).

[10] Eduardo H. Molina da Cruz, Marco A. Zanata Alves, Alexandre Carissimi, Philippe O. A. Navaux, Christiane. P. Ribeiro, and Jean-François Mehaut. 2011. Using Memory Access Traces to Map Threads and Data on Hierarchical Multi-core Platforms. In *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS'11*.

[11] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth. 2013. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'13*.

[12] Matthias Diener, Eduardo HM Cruz, Marco AZ Alves, Edson Borin, and Philippe OA Navaux. 2017. Optimizing memory affinity with a hybrid compiler/OS approach. In *Proceedings of the Computing Frontiers Conference*. ACM, 221–229.

[13] Matthias Diener, Eduardo H.M. Cruz, Laércio L. Pilla, Fabrice Dupros, and Philippe O.A. Navaux. 2015. Characterizing communication and page usage of parallel applications for thread and data mapping. *Performance Evaluation* 88-89 (2015), 18 – 36.

[14] Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Philippe O. A. Navaux, and Israel Koren. 2016. Affinity-Based Thread and Data Mapping in Shared Memory Systems. *ACM Comput. Surv.* 49, 4 (Dec. 2016).

[15] Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux, Anselm Busse, and Hans-Ulrich Heiß. 2014. kMAF: Automatic kernel-level management of thread and data affinity. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*.

[16] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. 2015. Challenges of Memory Management on Modern NUMA Systems. *Commun. ACM* 58, 12 (Nov. 2015), 59–66.

[17] Alfredo Giménez, Todd Gamblin, Barry Rountree, Abhinav Bhatele, Ilir Jusufi, Peer-Timo Bremer, and Bernd Hamann. 2014. Dissecting On-Node Memory Access Performance: A Semantic Approach. In *Proceedings of the conference on Supercomputing, Supercomputing'14*.

[18] Brice Goglin and Nathalie Furmento. 2009. Enabling high-performance memory migration for multithreaded applications on linux. In *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS'09*.

[19] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. 2012. MemProf: A Memory Profiler for NUMA Multicore Systems. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC'12*.

[20] Stefan Lankes, Boris Bierbaum, and Thomas Bemmerl. 2010. Affinity-On-Next-Touch: An Extension to the Linux Kernel for NUMA Architectures. In *Parallel Processing and Applied Mathematics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 576–585.

[21] Baptiste Lepers, Vivien Quema, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 277–289. https://www.usenix.org/conference/atc15/technical-session/presentation/lepers

[22] Xu Liu and John Mellor-Crummey. 2014. A Tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures. In *Proceedings of the symposium on Principles and Practices of Parallel Programming, PPoPP'14*.

[23] Xu Liu and Bo Wu. 2015. ScaAnalyzer: A Tool to Identify Memory Scalability Bottlenecks in Parallel Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*.

[24] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. PIN: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. 190–200.

[25] Jaydeep Marathe and Frank Mueller. 2006. Hardware Profile-guided Automatic Page Placement for ccNUMA Systems. In *Proceedings of the symposium on Principles and Practices of Parallel Programming, PPoPP'06*.

[26] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. 2010. Feedback-directed Page Placement for ccNUMA via Hardware-generated Memory Traces. *J. Parallel Distrib. Comput.* 70, 12 (Dec. 2010), 1204–1219. https://doi.org/10.1016/j.jpdc.2010.08.015

[27] C. McCurdy and J. Vetter. 2010. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software, ISPASS'10*.

[28] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. 2017. RTHMS: a tool for data placement on hybrid memory system. In *Proceedings of the International Symposium on Memory Management, ISMM'17*. 82–91.

[29] Manuel Selva, Lionel Morel, and Kevin Marquet. 2016. numap: A portable library for low-level memory profiling. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation, SAMOS'16*.

[30] Fengguang Song, Shirley Moore, and Jack Dongarra. 2007. Feedback-directed Thread Scheduling with Memory Considerations. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing, HPDC'07*.

[31] L. Zhu, H. Jin, and X. Liao. 2016. A Tool to Detect Performance Problems of Multi-threaded Programs on NUMA Systems. In *2016 IEEE Trustcom/BigDataSE/ISPA*.