Kernel-Based Thread and Data Mapping for Improved Memory Affinity

Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves,
Philippe O. A. Navaux, Anselm Busse, and Hans-Ulrich Heiss

Abstract—Reducing the cost of memory accesses, both in terms of performance and energy consumption, is a major challenge in shared-memory architectures. Modern systems have deep and complex memory hierarchies with multiple cache levels and memory controllers, leading to a Non-Uniform Memory Access (NUMA) behavior. In such systems, there are two ways to improve the memory affinity: First, by mapping threads that share data to cores with a shared cache, cache usage and communication performance are optimized. Second, by mapping memory pages to memory controllers that perform the most accesses to them and are not overloaded, the average cost of accesses is reduced. We call these two techniques thread mapping and data mapping, respectively. Thread and data mapping should be performed in an integrated way to achieve a compounding effect that results in higher improvements overall. Previous work in this area requires expensive tracing operations to perform the mapping, or require changes to the hardware or to the parallel application. In this paper, we propose kMAF, a mechanism that performs integrated thread and data mapping in the kernel. kMAF uses the page faults of parallel applications to characterize their memory access behavior and performs the mapping during the execution of the application based on the detected behavior. In an evaluation with a large set of parallel benchmarks executing on three NUMA architectures, kMAF achieved substantial performance and energy efficiency improvements, close to an Oracle-based mechanism and significantly higher than previous proposals.

Index Terms—Cache memories, shared memory, virtual memory, NUMA, memory affinity, thread mapping, data mapping

1 Introduction

The parallel computing landscape has experienced a large increase of thread-level parallelism in recent years as a result of the limits in the ILP scaling and on-chip power. Such highly parallel architectures lead to challenges for the memory accesses in shared-memory architectures, such as increased memory pressure and contention on the interconnections. These challenges have been met with a rising complexity of the memory hierarchy, consisting of multiple cache levels and multiple memory controllers per machine, which introduces a Non-Uniform Memory Access (NUMA) behavior for these machines [1]. As a result, deciding on which cores to execute the threads of parallel applications and on which NUMA nodes to place memory pages has a high impact on application performance and energy consumption [2], [3]. These two techniques to increase memory affinity are called *thread mapping* and *data mapping*, respectively.

Traditional mechanisms to improve performance in these architectures rely on mapping to increase the *locality* of memory accesses [4], [5]. By mapping threads that access shared data, which is also called communication, in such a way that they can communicate through shared caches, less cache misses and invalidations are expected [2], [5]. By mapping memory pages to NUMA nodes from which the most memory accesses are performed, the overall memory access latency is reduced, as well as the traffic between memory controllers [3], [5]. In addition to these locality-based techniques, recent research [6], [7] also discusses *balance* as an important factor to be considered when performing mapping, in order to avoid overloading some NUMA nodes. Taking both locality and balance into account when performing the mapping can result in higher improvements than when relying on a single metric only [7].

In this paper, we extend our recently-introduced *kernel Memory Affinity Framework (kMAF)* [5], which performs automatic thread and data mapping of parallel applications to improve their memory access behavior. It requires no changes to applications or prior information about their behavior. kMAF uses the page faults of a parallel application to determine its behavior, introducing extra faults to increase the detection accuracy. The detected behavior is analyzed to calculate optimized thread and data mappings and to perform migrations during the execution of the application.

kMAF was extended and improved in the following ways: (1) We ported kMAF to a new hardware architecture, Intel Itanium (IA64), which allows us to perform mapping on a traditional NUMA system. (2) kMAF now includes a new balance-based data mapping policy to distribute pages more fairly among NUMA nodes. (3) We include a new thread mapping mechanism that is more efficient and results in less unnecessary migrations than the previous. (4) We evaluate support for multiple parallel applications that are running at the same time. (5) We analyze kMAF's

- M. Diener, E. H. M. Cruz, M. A. Z. Alves, and P. O. A. Navaux are with the Informatics Institute of the Federal University of Rio Grande do Sul, Porto Alegre, Brazil.
E-mail: {mdiener, ehmcruz, mazalves, navaux}@inf.ufrgs.br.
- A. Busse and H.-U. Heiss are with the Communication and Operating Systems Group of the Technische Universität Berlin, Germany.
E-mail: {anselm.busse, hans-ulrich.heiss}@tu-berlin.de.

Manuscript received 20 May 2015; revised 18 Nov. 2015; accepted 22 Nov. 2015. Date of publication 3 Dec. 2015; date of current version 10 Aug. 2016.
Recommended for acceptance by U. V. Catalyurek.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2504985

sensitivity to the number of extra page faults. We also discuss the performance and energy consumption improvements of mapping using a simulator.

This paper is structured as follows: The next section discusses related work in the area of thread and data mapping. A detailed investigation of the benefits of thread and data mapping using a simulator is shown in Section 3. kMAF is introduced in Section 4. The evaluation methodology is shown in Section 5. Section 6 discusses the results of our experiments. Finally, Section 7 summarizes our conclusions.

2 Related Work

This section discusses previous research in the area of thread and data mapping.

2.1 Data Mapping

We divide previous research on data mapping into three broad groups according to where information about the memory accesses behavior is collected: OS-based mechanisms, compiler or runtime library-based mechanisms and mechanisms that operate on the hardware level.

2.1.1 OS-Based Mapping

Traditional data mapping strategies that have been employed by operating systems on NUMA architectures are *first-touch*, *interleave* and *next-touch*. In the first-touch policy [8], a page is allocated on the first NUMA node that performs an access to the page, and the page is never migrated. This is the default policy for most current operating systems, including Linux. In some circumstances, first-touch can lead to overloading of NUMA nodes, for example when a single thread initializes a large amount of data. In these cases, it can be beneficial to distribute memory pages more equally to balance the load on the memory controllers. The most common way to distribute pages is to use an interleave policy, which is available on Linux via the `numactl` tool [9]. In next-touch policies, a page is marked in such a way that it will be migrated to the node that performs the next access to it. Löf and Holmgren [10] propose such a mechanism for the Solaris operating system. In case a page is accessed by several nodes, this mechanism can lead to excessive migrations.

Modern improvements on the OS level focus mostly on refining the data mapping during execution, as the OS has no prior information about the application behavior. Recent versions of the Linux kernel include the NUMA Balancing technique [11] for the x86_64 architecture. NUMA Balancing uses the page faults of parallel applications to detect memory accesses and performs a sampled next-touch strategy. Whenever a page fault happens and the page is not located on the NUMA node that caused the fault, the page is migrated to that node. However, this mechanism keeps no history of accesses, which can lead to a high number of migrations, and it performs no thread mapping to improve the gains of the data mapping.

The Carrefour mechanism [6] has similar goals, but uses Instruction-Based Sampling (IBS), available on AMD architectures, to detect the memory access behavior and keeps a history of memory accesses to limit unnecessary migrations. Additionally, it allows replication of pages that are mostly read. Blagodurov et al. [12] discuss contention on NUMA architectures and identify thread migrations between NUMA nodes as detrimental to performance. Awasthi et al. [1] propose balancing the memory controller load by dynamically migrating pages from overloaded controllers to less loaded ones. They estimate load from row buffer hit rates gathered through simulation. Due to the runtime overhead, most of these mechanisms have to limit the number of samples they collect and/or the number of pages they characterize (30,000 pages in the case of Carrefour, for example). In Section 6, we compare Carrefour and NUMA Balancing to kMAF.

2.1.2 Compiler-Based and Runtime-Based Mapping

Other mechanisms perform mapping in user space, most of them on the compiler or runtime level. All these techniques have in common that they only have knowledge about a single application. Since systems execute several applications at the same time, their mapping decisions can interfere between them or with the OS, which limits their applicability to cases where execution of only a single application can be guaranteed. Some techniques, such as ForestGOMP [13], require source code annotations and are limited to specific parallelization libraries, such as OpenMP. Majo and Gross [14] identify memory accesses to remote NUMA nodes as a challenge for optimal performance and introduce a set of OpenMP directives to perform distribution of data. The best distribution policy has to be chosen manually and may differ between different hardware architectures. Piccoli et al. [15] use compiler-inserted code to predict memory access behavior in parallelized loops and use the prediction to migrate pages before the loop is executed. No thread mapping or balancing operations are performed. Nikolopoulos et al. [16] present an integrated compiler/OS-based data mapping mechanism based on a custom OpenMP compiler and IRIX kernel extensions. The compiler inserts instrumentation code to identify access patterns to shared memory areas, which are used to guide migration decisions.

Libraries that support NUMA-aware memory allocation include libnuma [9] and MAi [3]. With these libraries, data structures can be allocated according to the specification of the programmer, such as on a particular NUMA node, or with an interleave policy. These techniques can achieve large improvements, but place the burden of the mapping on the programmer and might require rewriting the code for each different architecture. An evolution of MAi, the Minas framework [17], optionally uses a source code preprocessor to determine data mapping policies for arrays. Previous research also uses memory access traces to perform data mapping [7], [18], [19]. These can be useful to determine the maximum gains that can be achieved with mapping policies, but are not applicable in general due to their substantial overhead and the fact that the access behavior might change with different input data and different numbers of threads, among others. We use such tracing techniques to implement an Oracle-based mapping in Sections 3 and 6.

2.1.3 Hardware-Based Mapping

Another category of mechanisms use statistics generated from hardware counters to guide mapping decisions. These techniques are generally limited to specific hardware architectures. Marathe and Mueller [20] present an automatic page placement scheme for NUMA platforms by tracking memory addresses from the performance monitoring unit (PMU) of the Intel Itanium processor. The profiling mechanism is enabled only during the start of each application due to the high overhead and therefore loses opportunities to handle changes during the rest of the execution. Tikir and Hollingsworth [21] use UltraSPARC III hardware counters to provide information for the data mapping, but do not perform thread mapping. Their proposal is limited to architectures with software-managed Translation Lookaside Buffers (TLBs), which covers only a minority of current systems. The Locality-Aware Page Table (LAPT) [22] is an extended page table that stores the memory access behavior for each page in the page table entry. This information is updated by the Memory Management Unit (MMU) on every TLB access, requiring hardware changes to the MMU. The operating system evaluates the behavior periodically to improve the thread and data mapping.

2.2 Thread Mapping

Thread mapping in shared-memory architectures is usually performed by detecting the communication between different threads, which is generally *implicit* as it is performed through memory accesses to shared memory regions. The detected behavior is fed to a mapping algorithm, which, together with information about the hardware topology, calculates an optimized thread mapping. Gains are limited on NUMA systems due to the lack of data mapping.

Autopin [23] measures the instructions per cycle of several mappings provided to it and executes the application with the thread mapping that presented the highest number of instructions. Radojkovic et al. [24] propose BlackBox, a scheduler that selects the best mapping by measuring the performance that each mapping obtained, similar to Autopin. When the number of threads is low, all possible thread mappings are evaluated. When the number of threads is too high to evaluate all possibilities, the authors execute the application with 1,000 random mappings to select the best.

Several mechanisms use indirect communication statistics from hardware counters. Azimi et al. [4] use hardware counters that provide memory addresses of requests resolved by remote cache memories. It detects incomplete communication patterns, since memory requests resolved by local caches are not considered. SPCD [25] uses page faults of parallel applications to detect communication. This mechanism keeps track of the threads that generate page faults and in which pages they happen, placing threads with lots of shared pages close to each other in the memory hierarchy.

3 Impacts of Thread and Data Mapping

To illustrate the impact of thread and data mapping on the performance and energy consumption, we analyze the mappings in a microarchitecture simulator. Different combinations of mappings are evaluated, generated with a mechanism based on memory access traces. The goal of this section is to provide an analysis of the sources of performance and energy consumption improvements from mapping, with statistics that real machines do not provide. We also discuss the impact of performing thread and data mapping jointly.

TABLE 1
Configuration of the Machine That was Simulated in SiNUCA

System: 4×2-core processors, L1 caches per core, L2 caches shared
NUMA: 4 mem. ctrl., NUMA fac.: 3, latency L2 to mem. ctrl.: 32–96 cyc.
Execution cores: OoO, 1.8 GHz, 65 nm, 12 stages, 16 byte fetch size
L1I/L1D caches: 32 KB, 8-way, 64 B line size, LRU policy, 1 cycle
Shared L2 caches: 512 KB, 8-way, 64 B line size, LRU policy
Interconnection: Bi-directional ring, hop latency: 32 cycles
DRAM: DDR2 667 MHz (5-5-5), 8 DRAM banks/channel, 2 channels

3.1 Simulation Methodology

We used SiNUCA [26], a performance-validated, cycle-accurate simulator for the Intel x86 architecture. It contains a detailed model of all architectural components and supports multi-core NUMA architectures. Table 1 presents the configuration parameters of the simulated machine. It contains four processors, each with its own memory controller, forming four NUMA nodes in total. The processors consist of two cores, with private L1 instruction/data caches and a shared L2 cache. The energy consumption of each component was estimated by feeding statistics generated by SiNUCA to the McPAT energy modeling tool [27].

For the evaluation, we selected the SP benchmark from the OpenMP implementation of the NAS Parallel Benchmarks [28], since it has a high sensitivity to data and thread mapping. SP was executed with eight threads and input size *W*. We generate memory access traces of SP with the Pin tool [29] to calculate the mappings.

In order to understand the importance of thread and data mapping and their impact on the performance and energy consumption, four different mapping configurations were simulated. For each configuration, we select either a *local* or *remote* policy for the thread and data mapping. In the local thread mapping (T_L), threads that access shared data are mapped close to each other in the hierarchy, while they are placed far apart in the remote thread mapping (T_R). Similarly, in the local data mapping (D_L), each memory page is mapped to the NUMA node that performs the most accesses to the page, while in the remote data mapping (D_R) each page is mapped to the node with the fewest accesses. This results in the following four combinations of configurations that are evaluated: $\mathbf{T_R D_R}$, $\mathbf{T_R D_L}$, $\mathbf{T_L D_R}$ and $\mathbf{T_L D_L}$.

3.2 Simulation Results

Fig. 1 presents the simulation results for the system components. The results are normalized to the values of the $T_R D_R$ mapping, which had the lowest performance.

We observe that using only the local data mapping ($T_R D_L$) improves the performance by 31 percent, providing higher benefits than performing only the local thread mapping ($T_L D_R$), which reduced execution time by 15 percent. The local data mapping ($T_R D_L$) improves the accesses to the main memory, by reducing the distance between the

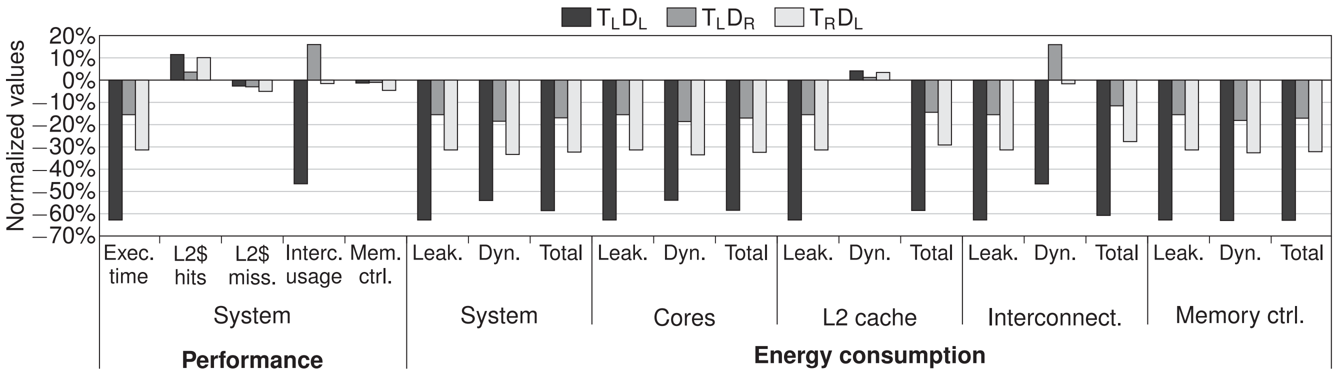


Fig. 1. Performance and energy consumption of the SP benchmark in SiNUCA, normalized to the remote thread and data mapping (T_RD_R).

threads and the data they access. Since these accesses are off-chip, they are high latency operations. On the other hand, the local thread mapping (T_LD_R) improves the usage of the shared cache memories (L2) by reducing the competition for the L2 cache space and improving the cache hit and miss ratios. As these are low-latency on-chip accesses, this mapping results in less improvements compared to the local data mapping.

When combining both local mappings (T_LD_L), the application performance improved by 62 percent, which is higher than the sum of the results from the local thread and data mappings applied separately. T_LD_L is able to reduce the number of cache misses and cache-to-cache transfers due to the local thread mapping, and also reduces the main memory average latency and interconnection contention due to the local data mapping. In addition, the local thread mapping also increases the gains from the local data mapping for pages that are shared between threads. By mapping threads that share data to the same NUMA node, main memory accesses from these threads will be considered as local, augmenting the benefits of the local data mapping [5].

Apart from performance, thread and data mapping can also improve the energy efficiency of parallel applications, for two main reasons. By reducing execution time, static energy consumption (leakage) will be reduced proportionally, since the processor is in a high power-consuming state for a shorter time. Less cache misses and traffic on the interconnections also reduce the dynamic energy consumption.

For all the evaluated mappings, the more efficient execution also reduced energy consumption. Leakage was reduced for all components, correlating with the execution time as expected. The main sources of dynamic energy savings were the reduction on the number of L1 cache misses, the interconnection traffic reduction and the reduction in the number of main memory accesses, since most of the accesses are being treated by the L2 cache memory. The overall energy reductions were slightly lower than the execution time reductions due to lower dynamic energy savings.

This section showed that substantial performance and energy improvements can be accomplished with mapping. The main benefits from the local thread and local data mapping can be observed on the memory sub-system and the interconnections, enabling a more efficient usage of resources. By combining both types of mappings, a compounding effect is achieved, resulting in higher improvements than when applying each technique separately. The Oracle-based mapping used in this section does not consider the difficulty or overhead to detect and perform improved mappings. Our proposed mechanism, kMAF, which will be discussed in the next section, aims to obtain similar improvements by performing an efficient online mapping.

4 The Kernel Memory Affinity Framework

The *kernel Memory Affinity Framework* uses the paging-based virtual memory implementation of modern computer architectures to analyze the memory access behavior of parallel applications during execution and uses the generated information to perform an improved thread and data mapping. This section presents kMAF, discusses its implementation and details its overhead.

4.1 Overview of KMAF

kMAF consists of four parts, which are shown in Fig. 2.

1) *Detect memory access behavior*. The memory access behavior of the parallel application is detected by tracing its page faults. Extra low-latency page faults are inserted throughout the execution to improve the detection accuracy.
2) *Storage and analysis of the behavior*. The memory addresses and thread IDs of the page faults are stored in two tables on the page and sub-page granularity, for data and thread mapping, respectively. For thread mapping, kMAF also maintains a sharing matrix.
3) *Perform thread mapping*. The sharing matrix is evaluated periodically with a mapping algorithm to determine if threads should be migrated between cores.
4) *Perform data mapping*. The access behavior to a page is analyzed during each page fault to determine if a page should be migrated between NUMA nodes.

4.2 Detecting Memory Access Behavior

Detecting the memory access behavior of parallel applications efficiently is a critical step of kMAF, since it determines the accuracy and overhead of the information and directly impacts the gains that can be achieved. As memory accesses are usually performed directly by the hardware, without a notification to the OS, gathering information about them in a portable way is challenging. In most architectures, the OS is notified about a memory access only when a page fault happens. We make use of

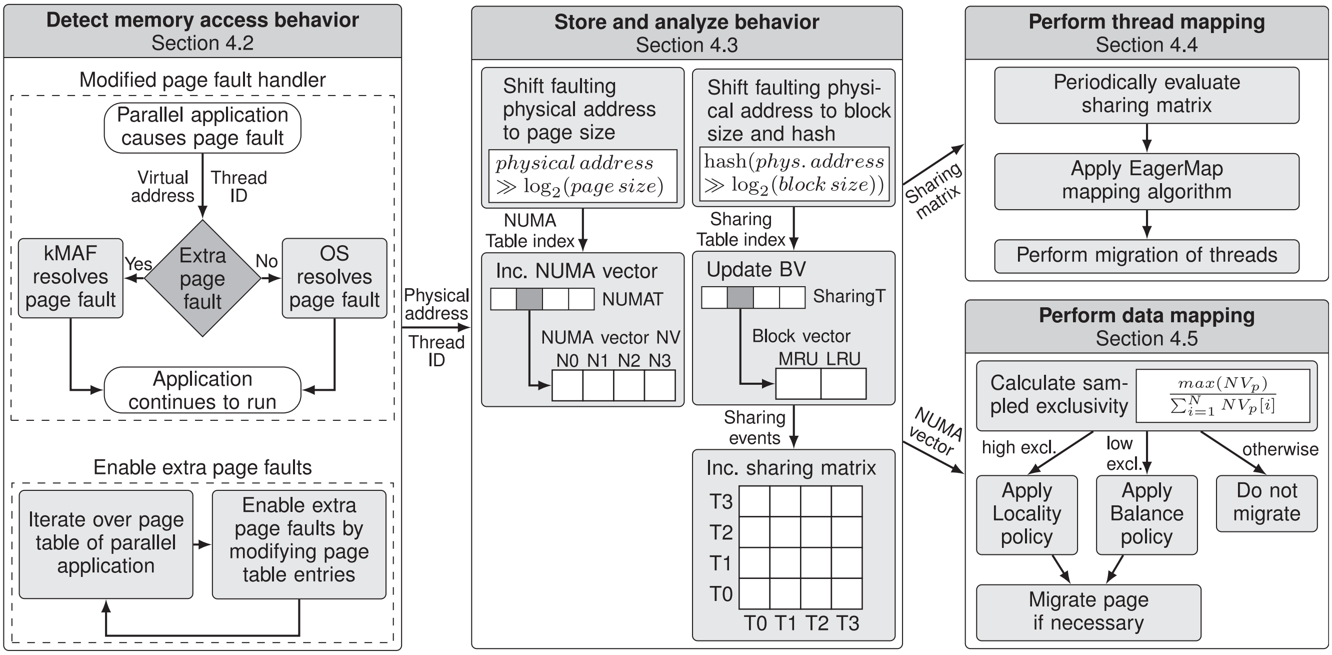

Detect memory access behavior
Section 4.2
Modified page fault handler
Parallel application causes page fault
Virtual address
Thread ID
Extra page fault
Yes
No
kMAF resolves page fault
OS resolves page fault
Application continues to run
Enable extra page faults
Iterate over page table of parallel application
Enable extra page faults by modifying page table entries
Physical address
Thread ID
Store and analyze behavior
Section 4.3
Shift faulting physical address to page size
$physical\ address \gg \log_2(page\ size)$
Shift faulting physical address to block size and hash
$hash(phys.\ address \gg \log_2(block\ size))$
NUMA Table index
Sharing Table index
Inc. NUMA vector
NUMAT
NUMA vector NV
N0 N1 N2 N3
Update BV
SharingT
Block vector
MRU LRU
Sharing events
Inc. sharing matrix
T3
T2
T1
T0
T0 T1 T2 T3
Sharing matrix
NUMA vector
Perform thread mapping
Section 4.4
Periodically evaluate sharing matrix
Apply EagerMap mapping algorithm
Perform migration of threads
Perform data mapping
Section 4.5
Calculate sampled exclusivity
$\frac{max(NV_p)}{\sum_{i=1}^{N} NV_p[i]}$
high excl.
low excl.
otherwise
Apply Locality policy
Apply Balance policy
Do not migrate
Migrate page if necessary

Fig. 2. Overview of the kMAF mechanism. Data structures are shown for a NUMA machine with four nodes (N0-N3) executing a parallel application consisting of four threads (T0-T3).

this fact in kMAF by observing the page faults of the parallel application.

4.2.1 Observing Memory Accesses via Page Faults

Whenever information about a virtual-to-physical address mapping is missing from the page table for a particular application, or when that information is invalid, the CPU signals a page fault to the operating system, including information about the faulting virtual address. The thread ID that caused the fault is determined by the OS. By tracking this information, kMAF can detect the memory access behavior.

Since the faulting address is the full address, not just the address of the page, different granularities can be applied for the detection, such as the cache line size or the specific page size of the architecture. This fact is important for the thread mapping due to its focus on improving the usage of caches. To support multiple running applications or processes that do not share a page table,[1] kMAF converts the virtual address to the physical address for the detection, which is unique in the system.

4.2.2 Increasing Detection Accuracy with Extra Page Faults

An important aspect of the detection is that in normal circumstances, only one page fault happens for each page. After the page fault has been resolved by the OS by inserting a translation into the page table, subsequent accesses to the page will not generate further page faults.[2] To overcome this restriction and to increase the accuracy of the detection, kMAF inserts extra page faults during the execution of a parallel application, such that multiple faults can happen for the same page. Page faults are inserted in all memory areas that store data, but not for code segments.

To insert the additional faults, kMAF periodically iterates over the page table of the parallel application and modifies entries in such a way that the next memory access to the page will generate another page fault. In most architectures, each page table entry contains a *present* bit which indicates if an entry is valid. To insert an extra page fault, kMAF clears this bit. As the extra page faults do not indicate missing information in the page table, no expensive operation (such as an allocation of a new page) has to be performed. The extra fault can be resolved by kMAF itself, by setting the present bit, without the use of other kernel routines. This reduces the overhead of these extra faults. To accurately characterize applications with different memory usages, kMAF scales the number of extra faults with the memory consumption of the application, as a percentage of the total of number pages that the application uses.

4.3 Storage and Analysis of the Detected Behavior

On each page fault, two pieces of information are gathered from the detection, the complete physical address that caused the fault and the thread ID. kMAF updates the memory access pattern with this data in two ways, on a per-page granularity for data mapping and on a finer granularity for thread mapping.

4.3.1 Data Mapping

Since data mapping operates on the page level, information is stored about the memory access pattern of each page from the NUMA nodes. For each page that is allocated (that is, which is accessed at least once), kMAF maintains a *NUMA vector (NV)*, where each element $NV_p[n]$ stores the

[1] Multithreaded applications running on Linux share a page table.
[2] Multiple page faults can happen for a single page in case multiple threads access the page for the first time in parallel.

number of page faults to page p from node n. On each page fault, the physical address is shifted to the page size with Equation (1),

$$index = physical\ address \gg \log_2(page\ size). \quad (1)$$

The resulting $index$ is used to access a *NUMA Table (NUMAT)* which stores the NUMA vectors for the pages. The NV_p of the page is then incremented for the NUMA node on which the thread that caused the page fault is executing.

4.3.2 Thread Mapping

For thread mapping, a different type of information is necessary. Thread mapping has the goal of optimizing the usage of the caches in the system by mapping threads that share data on cores that share caches, in addition to increasing the memory access locality for pages that are shared between threads [5]. For this reason, the page granularity, which is many times larger than the cache line size, is not sufficient to accurately detect the sharing behavior. Furthermore, since the thread mapping algorithm requires only an estimation of the amount of sharing between all pairs of threads (represented as a *sharing matrix*), storing information about every page is not necessary.

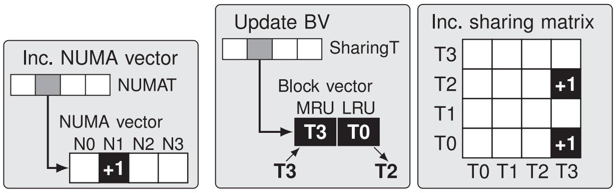
With these considerations, we implemented a simplified sharing detection for kMAF, based on our previous work [25], that drastically reduces the storage overhead compared to the previous version of kMAF [5]. We split the memory into *memory blocks* of a configurable size, with a default size of 256 bytes. The sharing detection is based on a hash table, which we call *sharing table*, that stores the IDs of the threads that recently accessed different memory blocks. The index of the hash table element is calculated with Equation (2). We use the default hash function provided by the Linux kernel,

$$index = \mathrm{hash}(physical\ address \gg \log_2(block\ size)). \quad (2)$$

For each block, kMAF stores a *block vector (BV)*, which contains a short list of the threads that previously accessed the block. In the default configuration shown in Fig. 2, the list has two elements, MRU and LRU. Whenever a thread accesses the block, its ID is inserted at the MRU position, shifting a previous ID to the LRU position. The element at the LRU position leaves the vector. kMAF increments the sharing matrix at the column of the thread ID and the rows of the previous threads that accessed the block.

4.3.3 Example of the Update of Data Structures

Fig. 3 contains an example of the update of the data structures during a page fault. Consider an architecture with four NUMA nodes, in which thread 3, executing on node 1, causes a page fault in a block that has been previously accessed by threads 0 and 2, with thread 0 performing the most recent access. kMAF then increments the NUMA vector of the page that was accessed in position 1, representing node 1 (Fig. 3a). The block vector of the block that was accessed contains two threads. Therefore, the oldest thread at the LRU position is removed from the BV (T2), T0 is shifted from the MRU to the LRU position, and T3 is inserted at the MRU position. Then, the sharing matrix is incremented for thread 3 with all the threads that were stored in the BV (Fig. 3b).



(a) Data mapping. (b) Thread mapping.

Fig. 3. Example of the update of data structures. Consider that thread 3 (executing on NUMA node 1) causes a page fault in a block that has been accessed by threads 0 and 2 before.

4.4 Thread Mapping

Since the number of threads of parallel applications in shared memory systems is currently relatively low (limited to several thousands of threads), it is feasible to evaluate the global communication behavior of the application to perform the thread mapping. The thread mapping problem is defined as finding a mapping of threads to processing units (PUs) that maximizes the locality, given a description of the communication behavior and the hardware hierarchy. Several algorithms have been suggested previously to calculate this mapping. Most require graph-based descriptions of the behavior and the hierarchy.

We selected the recently-proposed EagerMap algorithm [30] for kMAF, which is particularly suitable for online mapping mechanisms. Compared to previous algorithms, EagerMap has a very fast execution time (about 10 × faster than the second fastest, the Dual-Recursive Bipartitioning of the Scotch mapping library [31], with a similar scaling behavior). Despite the faster execution time, EagerMap achieves the same mapping quality as TreeMatch [32] and Scotch. Due to the greedy mapping strategy adopted by EagerMap, it has a high stability, which means that small changes in the communication behavior will have only small changes in the generated mapping. Since online communication detection implies that the communication behavior is constantly updated during execution, a high stability results in less unnecessary thread migrations, reducing the overhead of the thread mapping [30].

We ported EagerMap to the kernel as part of kMAF. The hardware topology (including information about caches and NUMA nodes) is generated directly from the information available in the kernel, without the need for external tools. During the execution of the parallel application, kMAF periodically fetches the sharing matrix and applies the EagerMap algorithm on the matrix and the hardware topology. The output of EagerMap is the PU where each thread should execute. kMAF then migrates threads to their assigned PUs.

4.5 Data Mapping

In contrast to the thread mapping, where the global communication was evaluated periodically, the memory access behavior to pages is evaluated locally (for each page) during every page fault. This is done for two reasons. As a parallel application can use millions of pages, performing the data mapping for all pages at the same time is not practical as it would lead to a substantial overhead for the calculation of the mapping and the page migrations. Furthermore, by analyzing the access behavior and performing eventual migrations during the page faults, there is no need for an additional context switch from the application to the kernel.

On the first access to a page, kMAF maintains the traditional first-touch semantics and allocates the page on the NUMA node that performs the first access to it. On subsequent accesses, the data mapping is performed in three steps during the page fault. First, the sampled exclusivity for the page is calculated from the NUMA vector, which describes if a page is (mostly) accessed from a single NUMA node [5]. The exclusivity is used to apply a locality-based or balance-based mapping policy to the page. The page is then migrated to the node returned by the mapping policy.

4.5.1 Sampled Exclusivity and Policy Selection

To determine which page migration policy should be a applied to a certain page p, we calculate the *sampled exclusivity* $Excl_{sample}$ of the page, as shown in Equation (3), where NV_p is the NUMA vector of page p, N is the total number of NUMA nodes in the system, and the max function returns the maximum value of its argument. Pages with a high exclusivity are accessed mostly from a single NUMA node and are therefore more suitable for locality-based mapping [5],

$$Excl_{sample}[p] = \frac{max(NV_p)}{\sum_{i=1}^{N} NV_p[i]}. \tag{3}$$

The mapping policy is then selected with Equation (4) based on the sampled exclusivity,

$$MapPolicy[p] = \begin{cases} Locality, & \text{if } Excl_{sample}[p] > min_{Loc} \\ Balance, & \text{if } Excl_{sample}[p] < max_{Bal} \\ none, & \text{otherwise.} \end{cases} \tag{4}$$

For pages with a high exclusivity ($> min_{Loc}$), which can benefit from increasing the locality of memory accesses, a Locality policy is applied. If the exclusivity is low ($< max_{Bal}$), kMAF applies a Balance policy, since this page can not benefit from a better locality. Otherwise, no mapping is performed and kMAF returns execution to the application. This process resembles our Oracle-based mapping mechanism that will be discussed in Section 5.3.

4.5.2 Locality Policy

If the Locality policy is selected, kMAF applies a mapping filter to reduce the number of page migrations in case the memory access behavior to the page changes quickly. This filter is expressed in Equation (5), where NV_p is the NUMA vector of page p, and the max / max_2 functions return the largest and second largest value of the vector, respectively,

$$Mig_{Local}[p] = \begin{cases} yes, & \text{if } max(NV_p) > 2 \times max_2(NV_p) + 1 \\ no, & \text{otherwise.} \end{cases} \tag{5}$$

The behavior of Equation (5) is illustrated in Table 2. For the illustration, consider that we are evaluating a single page p on a NUMA system with two NUMA nodes. Four steps (*a*–*d*) are shown in the table. Before the first step, page p has not been accessed and is not allocated on any node. On the first access to the page (step *a*), kMAF allocates it on node 1, which is the node that performed the access. The NUMA vector is incremented in position 1. After the page has been allocated, consider that subsequent accesses are performed from node 2. After the first (sampled) access from node 2 (step *b*), the NUMA vector is incremented, but no migration is performed, as the condition in Equation (5) is not fulfilled. It takes three more sampled accesses from node 2 for the page to be migrated to node 2 (step *c*). After the page has been migrated, consider that the access pattern changes again, with all accesses being performed from node 1. It now takes nine additional sampled accesses for the page to be migrated back to node 1 (step *d*). With this behavior, multiple migrations of the same page are becoming more difficult, limiting the number of migrations of pages with very frequent changes in the access behavior.

TABLE 2
Behavior of Equation (5) (Locality Data Mapping Policy) for a Single Page p and a NUMA Machine Consisting of 2 NUMA Nodes

Steps	NUMA vector		Sampled excl.	kMAF action
	N0	N1		
a) First acc. from N0	1	0	100%	Allocate p on N0
b) 1 acc. from N1	1	1	50%	None
c) 3 acc. from N1	1	4	80%	Migrate p to N1
d) 9 acc. from N0	10	4	71%	Migrate p to N0

Steps a – d *show the operation of kMAF for different memory accesses of the application to p, considering that the page has not been accessed before.*

The actual NUMA node where a page should be migrated to is calculated with Equation (6), where the $\arg\max$ function returns the element with the highest value,

$$node[p] = \arg\max(NV_p). \tag{6}$$

4.5.3 Balance Policy

kMAF limits the amount of migrations between the Balance and Locality policies by only applying the Balance policy after the access pattern has stabilized, as expressed by Equation (7), where N is the number of NUMA nodes,

$$Mig_{Balance}[p] = \begin{cases} yes, & \text{if } sum(NVp) > N \\ no, & \text{otherwise.} \end{cases} \tag{7}$$

If this condition is fulfilled, kMAF balances the page by using an *Interleave* policy, which has shown good balancing results [7]. The NUMA node for a page is calculated with the help of the address of a page, as shown in Equation (8), where $addr(p)$ is the virtual address of page p,

$$node[p] = (addr(p) \gg \log_2(page\ size)) \bmod N. \tag{8}$$

Such an Interleave policy has two main advantages compared to more complex techniques: (1) Since the NUMA node is calculated directly from the page address, there is no need to store or iterate over the global state of the application (such as the NUMA nodes or NUMA vector of all

TABLE 3
Default Configuration of kMAF Used in the Experiments

kMAF part	Configuration
Thread mapping	Hash table: 1 million blocks, each 256 byte large Mapping interval: 100 ms
Data mapping	min_{Loc}: 80%; $maxBal$: $1/N \times 1.5 \times 100\%$
Extra page faults	10% of total pages/second

pages) to determine the node to migrate to, reducing the overhead to calculate the mapping. (2) For the same reason, a page with a low exclusivity is migrated only once as long as its exclusivity remains low, reducing the number of unnecessary page migrations.

4.5.4 Performing the Page Migration

To perform the page migration, kMAF needs the virtual address of the page, as well as the NUMA node where it should be migrated to. Before migrating, kMAF checks if the page is not already located on the node to be migrated to, and aborts the migration in that case. Otherwise, kMAF uses the `unmap_and_move()` function of the Linux kernel to perform the actual page migration to the requested NUMA node. The virtual address is not stored by kMAF, since it is available during the page fault and page migrations are only performed while the fault is handled.

4.6 Supporting Multiple Running Applications

One important advantage of kernel-based mapping solutions such as kMAF is their support for multiple applications that execute concurrently, in contrast to many user-space techniques. Since kMAF bases its detection of the memory access behavior on the physical addresses of the memory accesses, different applications executing at the same time do not interfere with each other, maintaining the same detection accuracy as if only one application was running. For the thread mapping, kMAF maintains a single sharing matrix for the applications running, and applies the thread mapping algorithm with this matrix. The data mapping is performed without changes for multiple applications. In this way, multiple parallel applications are handled as one larger application and neither kMAF nor its configuration needs to be changed to support this case.

4.7 Implementation of KMAF

We implemented kMAF as a module for the Linux kernel. Compared to a kernel patch, a module has the advantage of not requiring a reboot to install kMAF. The default page fault of the kernel was modified to enable the page fault tracking mechanism and to implement the data mapping. The kMAF module creates a kernel thread that enables extra page faults during execution. Another kernel thread implements the thread mapping mechanism. Table 3 contains an overview of the default configuration parameters of kMAF used in our experiments.

4.8 Overhead of KMAF

Since kMAF operates during the execution of parallel applications, it imposes a storage and execution time overhead.

4.8.1 Storage Overhead

kMAF needs to allocate memory for the NUMA table, sharing table and sharing matrix. The size of the NUMA table is calculated with Equation (9), where N is the number of NUMA nodes in the system and P is the number of pages that the application accesses. We store the NUMA vector with an element size of 1 byte, which can count up to 256 page faults per node and per page. Considering a system that consists of four NUMA nodes and a page size of 4 KByte, the storage overhead per page is $4/4,096 \approx 0.1\%$,

$$size(NUMAT) = N \times P \times 1\ byte. \tag{9}$$

For the hash table that stores the sharing behavior, we store two thread IDs per block. Each ID has a size of 2 bytes to support up to 65,536 threads. In case of a hash conflict, the old block is overwritten. This method has shown a good detection accuracy [25]. Equation (10) shows the overhead for the hash table, where $nBlocks$ is the number of blocks that can be stored in it. In the default configuration with 1 million blocks, it has a size of 4 MByte,

$$size(SharingT) = 2 \times 2\ byte \times nBlocks. \tag{10}$$

The sharing matrix used to calculate the thread mapping has a cell size of 4 byte. The total size of the matrix can then be calculated with Equation (11), where T is the number of threads of the parallel application. For an application with 1,024 threads, the matrix has a size of 4 MByte,

$$size(SharingMatrix) = T^2 \times 4\ byte. \tag{11}$$

4.8.2 Execution Time Overhead

The runtime overhead consists of the time required to introduce extra page faults, resolve these faults, calculate the thread and data mappings, and the migrations. The complexity to introduce extra page faults increases linearly with the memory usage of the application. Resolving an extra page fault from kMAF has a constant time complexity. Our thread mapping algorithm has a complexity of $\mathcal{O}(T^3)$, where T is the number of threads of the parallel application. For data mapping, the time complexity is $\mathcal{O}(N)$, where N is the number of NUMA nodes. In Section 6, we will evaluate the runtime overhead on a running application.

5 Methodology of the Evaluation

We evaluate and compare kMAF on three machines using two parallel benchmark suites.

5.1 Hardware Architectures

We execute the benchmarks on three real NUMA machines: *Itanium*, *Xeon* and *Opteron*.

The *Itanium* machine represents a traditional NUMA architecture based on the SGI Altix 450 platform [33]. It consists of two NUMA nodes, each with two dual-core Intel Itanium 2 processors (Montecito microarchitecture [34]) and a proprietary SGI interconnection, NUMAlink [33]. Each core has private L1, L2 and L3 caches.

The *Xeon* machine represents a newer generation NUMA system with high-speed interconnections. It consists of

TABLE 4
Overview of the Three Systems Used in the Evaluation

Name	Property	Value
Itanium	NUMA	2 nodes, 2 processors/node, NUMA factor 2.1
	Processors	4×Intel Itanium 2 9030, 1.6 GHz, 2 cores
	Caches	16 KB+16 KB L1, 256 KB L2, 4 MB L3
	Memory	16 GB DDR-400, page size 16 KB
Xeon	NUMA	4 nodes, 1 processor/node, NUMA factor 1.5
	Processors	4×Intel Xeon X7550, 2.0 GHz, 8 cores, SMT
	Caches	8×32 KB+32 KB L1, 8×256 KB L2, 18 MB L3
	Memory	128 GB DDR3-1066, page size 4 KB
Opteron	NUMA	8 nodes, 2 nodes/processor, NUMA factor 2.8
	Processors	4×AMD Opteron 6386, 2.8 GHz, 8 cores, SMT
	Caches	8×16 KB+64 KB L1, 8×2 MB L2, 2×6 MB L3
	Memory	128 GB DDR3-1600, page size 4 KB

four NUMA nodes with 1 eight-core Intel Xeon processor each (Nehalem-EX microarchitecture [35]) and a QuickPath Interconnect (QPI) interconnection [36] between the nodes. Each core has private L1 and L2 caches, while the large L3 cache is shared among all the cores in the processor.

The *Opteron* machine represents a recently introduced generation of NUMA systems with multiple on-chip memory controllers. It consists of four AMD Opteron processors (Abu Dhabi microarchitecture [37]), each with two memory controllers, forming eight NUMA nodes in total. Each core has a private L1 data cache, while the L1 instruction cache and L2 cache is shared between pairs of cores. The L3 cache is shared among eight cores in the same processor.

The Itanium machine can execute up to eight threads concurrently, while *Xeon* and *Opteron* can execute 64 threads concurrently. The NUMA factors of the machines, which represent the overhead of memory accesses to remote NUMA nodes compared to the local node, were measured with Lmbench [38]. A summary of the machines, including their NUMA factors, is shown in Table 4.

5.2 Benchmarks

We used two parallel benchmark suites for our evaluation:

NAS-OMP [28] is the OpenMP implementation of the NAS Parallel Benchmarks (NPB). NAS-OMP consists of 10 applications from the High-Performance Computing domain. We used the *C* input size for the benchmarks on *Xeon* and *Opteron*, which represents a medium-large input size. The DC benchmark was executed with its largest input size, *B*. Since the *Itanium* machine is considerably smaller than the others, we execute the *B* input size for all NAS-OMP benchmarks on this machine. The average memory consumption per application is 6.1 GByte (*C* input size).

The *PARSEC* benchmark suite [39] is a set of 13 parallel applications for modern multi-core architectures, implemented in Pthreads and OpenMP. PARSEC was executed with its largest input size, *native*, on all three machines. The average memory usage per application is 3.2 GByte.

The benchmarks were executed with the maximum number of threads that each machine can execute in parallel: 8 threads on *Itanium* and 64 threads on *Xeon*/*Opteron*. Some PARSEC benchmarks cannot be executed with this number of threads and use numbers of threads that are as close as possible to 8/64. We show the average results of 10 executions for each configuration and the standard deviation.

5.3 Mapping Mechanisms

The following mapping mechanisms were compared:

OS. The Linux OS forms the baseline for our experiments. We run an unmodified Linux kernel, version 3.13, and use its default first-touch mapping policy. The NUMA Balancing mechanism [11] is disabled in this configuration.

Compact. The compact thread mapping is a simple mechanism to improve memory affinity by placing threads with neighboring IDs (such as threads 0 and 1) close to each other in the memory hierarchy, such as on same cores, similar to options available in some OpenMP environments [40].

Oracle. To calculate an oracle-based thread and data mapping, we use a memory tracer based on the Pin Dynamic Binary Instrumentation (DBI) tool [29]. For the data mapping, we map pages with a high exclusivity to the NUMA node with the most accesses to them, and shared pages with an interleave policy. This mapping results in the highest improvements [7]. We calculate a thread mapping with the EagerMap algorithm, as in kMAF. The generated mappings are stored in files. These files are read by a custom Linux kernel during application startup, which then performs the thread and data mapping. Since no migrations are performed and all calculations are done before the application starts, this mechanism has no runtime overhead.

NUMA Balancing. We use the NUMA Balancing mechanism [11] of version 3.13 of the Linux kernel. NUMA Balancing is only supported on the *Xeon* and *Opteron* machines.

Carrefour. The Carrefour mechanism [6] was evaluated on the *Opteron* machine, since it requires hardware features that are available only on AMD architectures. Carrefour was executed with its default configuration.

kMAF. Our proposed mechanism, kMAF, was implemented as a module for the Linux kernel (version 3.13) and executed with the configuration presented in Section 4.7.

6 Results

This section presents the experimental results of kMAF. We begin with the results of a single parallel application and multiple applications that are executing concurrently. We then evaluate the sensitivity of kMAF to the amount of extra page faults and discuss its runtime overhead.

6.1 Single Applications

The results for the three machines when executing a single parallel application at a time are shown in Figs. 4, 5, 6.

The results for the *Itanium* machine are shown in Fig. 4. Most of the NAS-OMP benchmarks benefit from an improved mapping, indicated by the substantial improvements compared to the OS. The highest improvements were achieved for the CG benchmark, of up to 65 percent with the Oracle policy. Due to the relatively low number of NUMA nodes (2) and the simple memory access pattern of most NAS-OMP benchmarks, performing only the Compact thread mapping already results in high speedups. For the majority of the NAS-OMP benchmarks, the results of kMAF are between the Compact and Oracle policies. Only two of the PARSEC benchmarks, Facesim and X264, are suitable for mapping on this architecture, with similar performance improvements for kMAF and the Oracle. For Ferret and X264, the Compact policy results in substantial performance

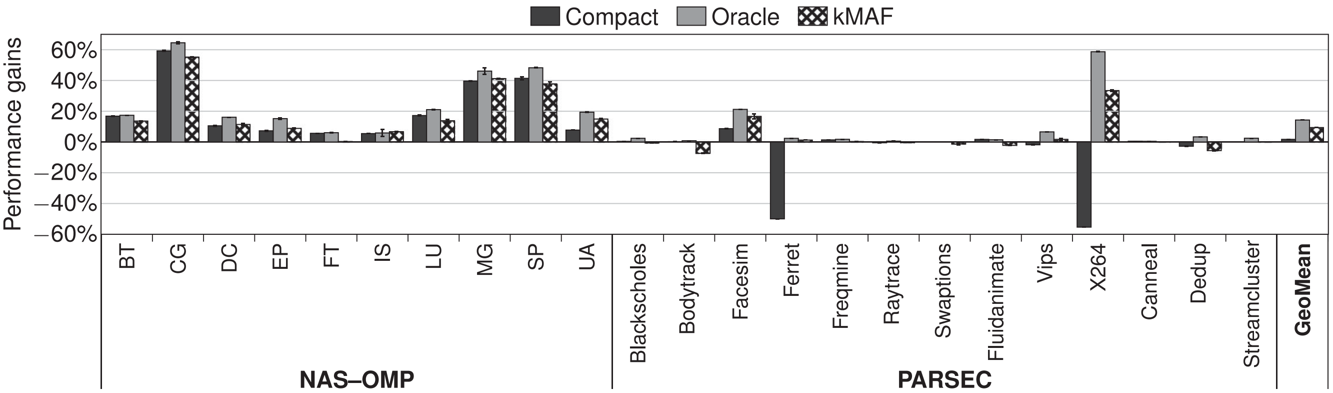


Fig. 4. Performance improvements on *Itanium*, normalized to the results of the OS.

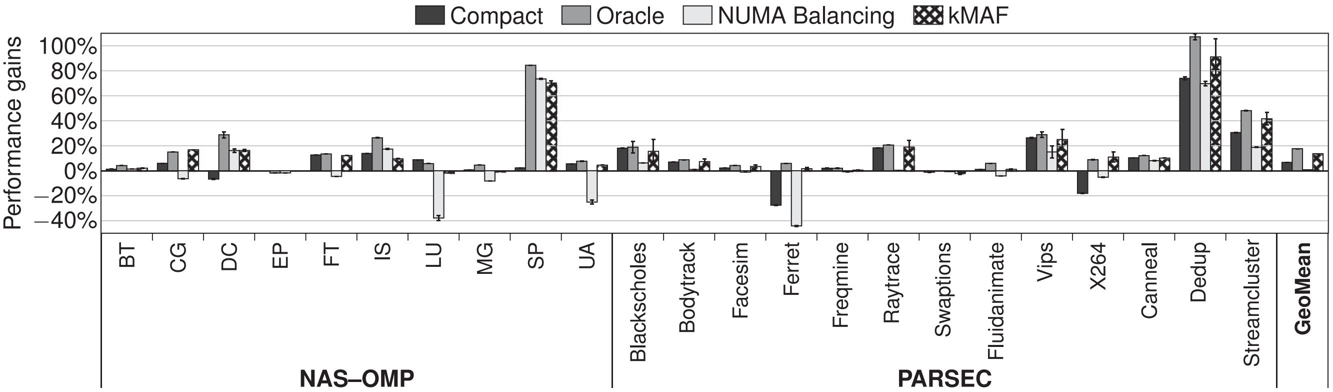


Fig. 5. Performance improvements on *Xeon*, normalized to the results of the OS.

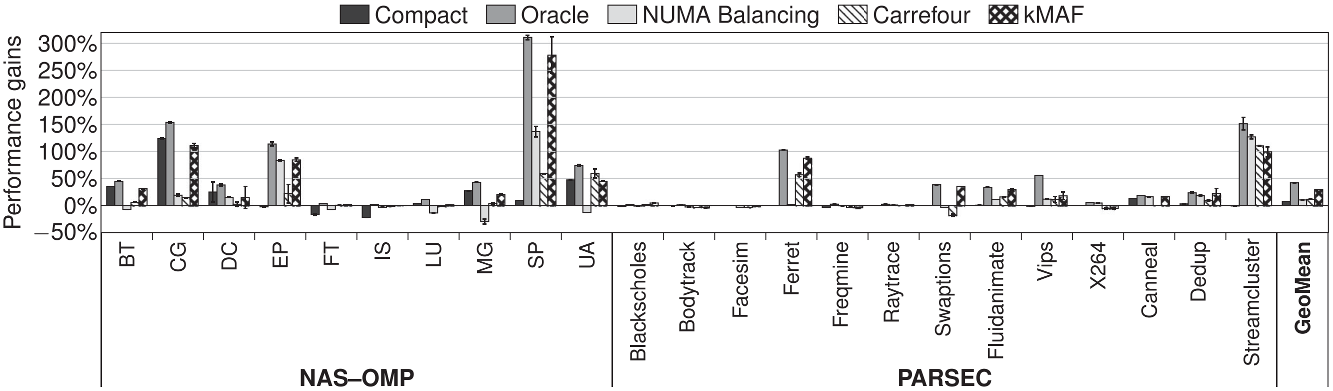


Fig. 6. Performance improvements on *Opteron*, normalized to the results of the OS.

losses compared to the OS. The geometric mean of the improvements of all benchmarks is 1.7, 14.3, and 9.4 percent for Compact, Oracle, and kMAF, respectively.

Fig. 5 shows the performance results for *Xeon*. From the NAS-OMP benchmarks, the highest improvements were achieved for the SP benchmark, with similar improvements for the Oracle, NUMA Balancing and kMAF. Despite the good improvements for some benchmarks (DC, IS and SP), NUMA Balancing causes significant slowdowns for others that can not benefit from mapping (CG, LU, MG and UA). The lack of an access history causes unnecessary page migrations and increases the runtime overhead. The Compact thread mapping only has minimal improvements, indicating the importance of data mapping. Several PARSEC benchmarks benefit from mapping on *Xeon*. Dedup's performance was improved by up to 100 percent. Due to their lower memory usage, many PARSEC applications can benefit from thread mapping only, as evidenced by the results of the Compact mapping. For most PARSEC benchmarks, kMAF has the closest results to the Oracle of all policies. The geometric mean of the improvements of all benchmarks is 6.7, 17.7, 0.8, and 13.6 percent for Compact, Oracle, NUMA Balancing, and kMAF, respectively.

The results for the *Opteron* machine are shown in Fig. 6. Overall, the highest performance improvements of the three machines were achieved on *Opteron*, due to its many NUMA nodes and high NUMA factor. Similar to *Xeon*, SP achieved the highest gains of the NAS-OMP benchmarks, of nearly 300 percent with kMAF. As before, kMAF had the closest results to the Oracle. Due to the high memory usage of the applications, the Carrefour mechanism had only low improvements since it limits itself to 30,000 pages

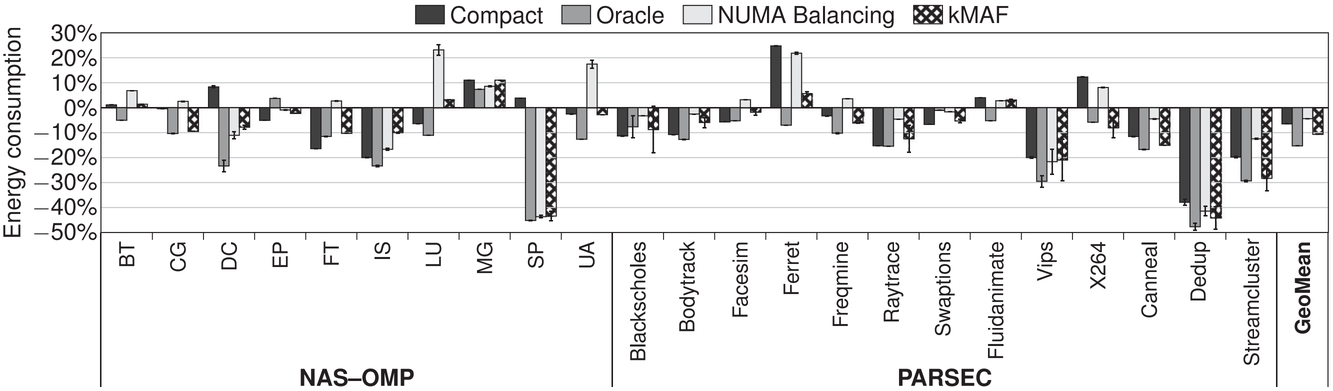


Fig. 7. Energy consumption results on *Xeon*, normalized to the OS.

(corresponding to 120 MByte with 4 KByte pages). The Compact thread mapping had negligible improvements except for CG. For the PARSEC benchmarks, Carrefour shows improvements that are closer to kMAF and the Oracle, since many PARSEC applications have a lower memory consumption (about 110 MByte in the case of Streamcluster, for example). However, both NUMA Balancing and kMAF have higher gains for most PARSEC benchmarks. The geometric mean of the improvements of all benchmarks is 7.7, 42.0, 11.6, 10.4, and 29.8 percent, for Compact, Oracle, Carrefour, NUMA Balancing, and kMAF, respectively.

Results show that simple ways to improve memory affinity, such as the Compact thread mapping, do not result in significant gains compared to the OS in most cases. Furthermore, mechanisms that keep no access history (NUMA Balancing) or limit the number of pages that they characterize (Carrefour) also do not result in optimal performance. Our kMAF mechanism provided the highest improvements overall, with substantial gains compared to the OS, close to the Oracle mechanism, on all three machines.

6.2 Energy Consumption

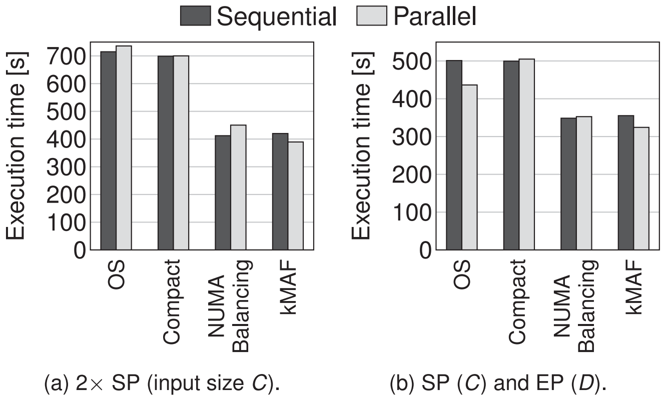
As discussed in Section 3.2, improved mappings can also result in higher energy efficiency. We evaluate the energy consumption improvements of the *Xeon* machine by using the Baseboard Management Controller (BMC), which exposes the energy consumption of the whole system through IPMI. The experimental methodology is the same as before. The energy consumption, normalized to the results of the OS, is shown in Fig. 7. As expected, the benefits are similar to the performance improvements, with applications that benefit more from mapping having higher energy savings. The highest improvements (of about 50 percent) were achieved for the SP and Dedup benchmarks. Similar to the performance results, the Compact thread mapping and NUMA Balancing did not achieve consistent improvements and actually reduce energy efficiency in several cases.

6.3 Multiple Applications

An important feature of kMAF is that it seamlessly supports multiple parallel applications that are executing concurrently, as discussed in Section 4.6, in contrast to solutions that operate in user space. We evaluate the support for multiple applications in this section.

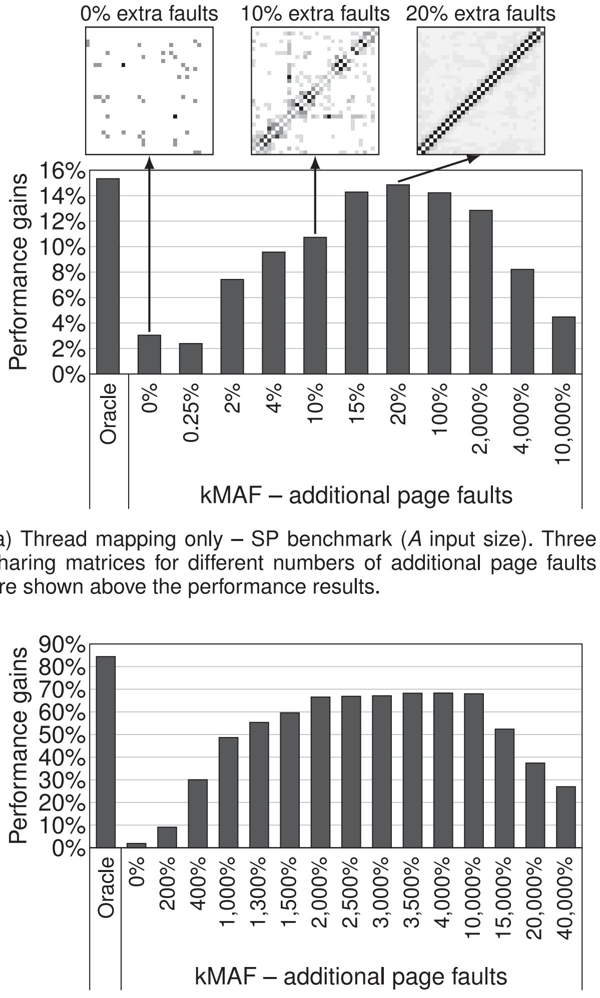
We selected two pairs of the parallel applications, SP+SP, and SP+EP. SP is executed with the *C* input size, as before, but we use the *D* input size for EP to achieve comparable execution times. All applications were executed with 64 threads each on the *Xeon* machine. Four mapping mechanisms (OS, Compact, NUMA Balancing and kMAF) and two different configurations (Sequential and Parallel) were compared. In the sequential configuration, the second application starts after the first one terminates. In this configuration, there is less interference between applications and less contention for resources such as caches, interconnections, memory controllers and functional units. However, the utilization of resources may not be optimal in this case. For example, when a thread stalls while waiting for a memory request, another thread might make use of functional units at that time. In the parallel configuration, both applications are started at the same time, and we measure the time until both terminate. This configuration has the opposite characteristics of the sequential case, with higher contention for resources, but potentially more efficient usage.

Fig. 8 shows the execution time of the experiment. In the SP+SP case (Fig. 8a), two memory-hungry applications are running, which leads to performance reductions for the OS and NUMA Balancing in the parallel configuration. Since kMAF additionally performs thread mapping, it is able to reduce the overall contention and gains performance from the parallel execution as well. In the SP+EP case (Fig. 8b), since EP is a CPU-bound application, it only competes for the functional units, not for memory accesses. For this reason, there is less overall contention, and even the OS can



(a) 2× SP (input size *C*). (b) SP (*C*) and EP (*D*).

Fig. 8. Running multiple applications on the *Xeon* machine. In the *sequential* configuration, applications run one after the other, and the total execution time is shown. In the *parallel* configuration, both applications start at the same time, and the execution time until both finish is shown.



(a) Thread mapping only – SP benchmark (*A* input size). Three sharing matrices for different numbers of additional page faults are shown above the performance results.

(b) Thread and data mapping – SP benchmark (*C* input size).

Fig. 9. Performance improvements of kMAF when varying the number of additional page faults on the *Xeon* machine. All values are normalized to the OS. The percentages on the *x*-axis indicate the number of additional page faults compared to the baseline without additional faults.

benefit from parallel execution, in addition to kMAF. The results for NUMA Balancing show that performing only data mapping is not sufficient for optimal results since the thread mapping can help to reduce contention.

6.4 Mechanism Sensitivity

kMAF's improvements are sensitive to the number of page faults. Increasing the number of extra faults increases the detection accuracy, but also increases the overhead on the running application. This section evaluates this impact.

As mentioned in Section 4.3.2, thread mapping requires less information than data mapping, as only information about the sharing pattern is needed, without knowing which data actually gets accessed. We execute two experiments with kMAF, one with thread mapping only, one with both thread and data mapping, to evaluate the different impacts. We selected the SP benchmark from NAS-OMP, since it has the most illustrative behavior, and execute kMAF with various numbers of additional page faults on the *Xeon* machine, comparing the results to the Oracle mapping and the OS.

The performance results with thread mapping only are shown in Fig. 9a. Values are normalized to the OS. On the horizontal axis, we show the improvements of the Oracle and the improvements of kMAF with various numbers of extra page faults, given as a percentage of the original number of faults. The figure also shows the sharing matrices for three different percentages of extra faults, 0, 10, and 20 percent. Even for 0 percent of extra faults, kMAF improves performance slightly compared to the OS, despite the inconclusive sharing matrix. The reason is that kMAF reduces the number of unnecessary thread migrations compared to the OS.

When increasing the number of extra faults, the gains from kMAF are rising as well. At 10 percent of extra faults, the communication pattern already becomes visible, with large amounts of communication between neighboring threads. At 20 percent of extra faults, kMAF reaches its maximum improvements, which are very close to the Oracle. Increasing the faults further does not improve the performance gains, and the overhead begins to affect the application. However, even with 10,000 percent of extra faults (corresponding to extra 100 faults per page), performance is still higher than the OS. This experiment shows that even with very little information about the application behavior (less than 1 extra fault per page), thread mapping can be performed effectively.

Data mapping requires information about every page, demanding several page faults per page to perform a successful mapping. Fig. 9b shows the results of kMAF when varying the number of extra faults. The result show that even with 200 percent extra faults, only small improvements can be achieved. The gains reach a maximum at 2,000% – 4,000% extra faults, becoming close to the Oracle mechanism. The default configuration of kMAF used in this paper generates about 2,000 percent extra faults. Increasing the number of faults beyond 4,000 percent reduces the application performance due to the increasing overhead. Even with 40,000 percent extra faults, performance is still substantially higher than the OS.

6.5 Data Mapping with Larger Pages

Large pages present challenges for data mapping, since the granularity of migration decisions is increasing. However, large pages result in a more efficient execution due to fewer page faults and TLB misses. We execute the SP benchmark on the *Xeon* machine with the OS mapping and kMAF with two page sizes, small (4 KByte) and large pages (2 MByte). Both cases use the same kMAF configuration.

The results of this experiment are shown in Fig. 10. By only enabling large pages and letting the OS handle thread and data mapping, performance is already improved by 36 percent. However, improvements are still higher with kMAF, even with small pages (71 percent). Running kMAF with large pages only increases performance slightly, to 89 percent compared to the OS with small pages and 39 percent with the large pages. With the larger page size, TLB misses are reduced by about 60 percent and page faults by 50 percent, with the OS mapping. These results confirm our intuition that larger pages reduce the impact of data mapping, reducing the improvements by half. Nevertheless, data mapping still remains important.

6.6 Overhead of KMAF

Since it operates during the execution of parallel applications, kMAF imposes a runtime overhead. Fig. 11 shows the number of execution cycles of a single event of each type of

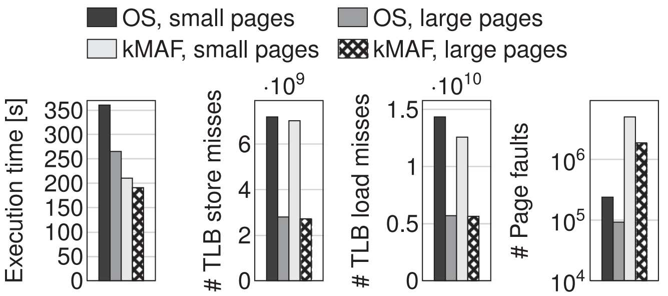


Fig. 10. Performance results of SP when running with small (4 KByte) and large pages (2 MByte) on the *Xeon* machine.

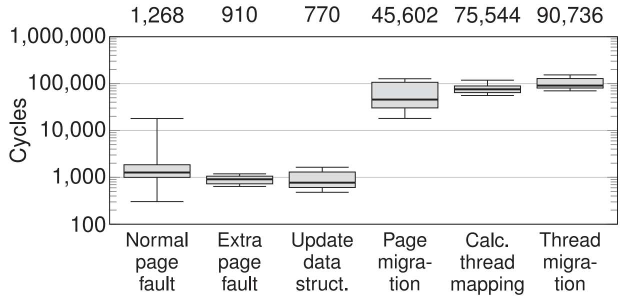


Fig. 11. Number of cycles of each kMAF operation on the *Xeon* machine. Numbers on the top show the median value for each type.

operation that kMAF performs, measured using the time stamp counter (TSC) on the *Xeon* machine. For each type of operation, the box plot shows the maximum, upper quartile, median, lower quartile, and minimum value. The median values are also shown above each category. We can confirm that the overhead of an extra page fault, which consists of the time to create and resolve it, is significantly lower than the overhead of the normal faults. In absolute terms, performing the page and thread migration and calculating the thread mapping have the highest computational demand. However, these are infrequent operations.

To evaluate the combined overhead during the execution of an application, we measure the total overhead of each category when executing the SP benchmark, which had the highest overhead in our experiments. Table 5 contains the number of events for each event type, absolute time spent for each event, and the overhead in % of the total execution time of SP. The results show that the overhead on the application is dominated by three categories, the extra page faults, the update of data structures and the page migrations. The total overhead corresponds to 2.4 percent of the total execution time, considerably lower than in the previous version of kMAF [5], which resulted in an overhead of 3.6 percent.

TABLE 5
Overhead of the SP Benchmark, Measured on the *Xeon* Machine

Value	Normal page faults	Extra page faults	Update data struct.	Page migr.	Calculate thread mapping	Thread migr.
# events	235,520	4,828,171	5,063,691	176,194	3,487	370
Abs. time	149 ms	2.19 s	1.95 s	4.01 s	130 ms	17 ms
% of total	0.04%	0.63%	0.56%	1.15%	0.04%	0.01%

7 Conclusions

Improving the memory affinity of parallel applications via optimized thread and data mappings is the key to improve their performance and energy consumption on shared-memory architectures. In this paper, we proposed kMAF, a kernel-based framework to improve affinity. kMAF uses page faults of parallel applications to determine their memory access behavior and to optimize the mapping online. In an evaluation with two parallel benchmark suites on three different NUMA systems, kMAF showed substantial performance and energy efficiency improvements with a low overhead, with gains close to an Oracle mechanism. For the future, we will evaluate the impact of the hierarchical structure of modern NUMA systems on the data mapping.

Acknowledgments

This research was partly supported by CNPq and CAPES.

References

[1] 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 *Proc. 9th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 319–330.

[2] W. Wang, T. Dey, J. Mars, L. Tang, J. W. Davidson, and M. L. Soffa, "Performance analysis of thread mappings with a holistic view of the hardware resources," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, 2012, pp. 156–167.

[3] C. P. Ribeiro, J.-F. Mehaut, A. Carissimi, M. Castro, and L. G. Fernandes, "Memory affinity for hierarchical shared memory multiprocessors," in *Proc. Int. Symp. Comput. Archit. High Perform. Comput.*, 2009, pp. 59–66.

[4] R. Azimi, D. K. Tam, L. Soares, and M. Stumm, "Enhancing operating system support for multicore processors by using hardware performance monitoring," *ACM SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 56–65, 2009.

[5] 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 *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2014, pp. 277–288.

[6] 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 *Proc. 18th Int. Conf. Archit. Support Programm. Lang. Oper. Syst.*, 2013, pp. 381–394.

[7] M. Diener, E. H. M. Cruz, and P. O. A. Navaux, "Locality vs. Balance: Exploring data mapping policies on NUMA systems," in *Proc. Int. Conf. Parallel, Distrib. Netw.-Based Process.*, 2015, pp. 9–16.

[8] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. Scott, "Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems," in *Proc. Int. Parallel Process. Symp.*, 1995, pp. 480–485.

[9] A. Kleen. (2004). An NUMA API for Linux. http://andikleen.de/numaapi3.pdf

[10] H. Löf and S. Holmgren, "Affinity-on-next-touch: Increasing the performance of an industrial PDE solver on a cc-NUMA system," in *Proc. Int. Conf. Supercomput.*, 2005, pp. 387–392.

[11] J. Corbet. (2012). Toward better NUMA scheduling [Online]. Available: http://lwn.net/Articles/486858/

[12] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A case for NUMA-aware contention management on multicore systems," in *Proc. USENIX Annu. Tech. Conf.*, 2010, pp. 557–571.

[13] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst, "Structuring the execution of OpenMP applications for multicore architectures," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2010, pp. 1–10.

[14] Z. Majo and T. R. Gross, "Matching memory access patterns and data placement for NUMA systems," in *Proc. Int. Symp. Code Generation Optim.*, 2012, pp. 230–241.

[15] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira, "Compiler support for selective page migration in NUMA architectures," in *Proc. Int. Conf. Parallel Archit. Compilation*, 2014, pp. 369–380.

[16] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé, "User-level dynamic page migration for multiprogrammed Shared-memory multiprocessors," in *Proc. Int. Conf. Parallel Process.*, 2000, p. 95.

[17] C. Ribeiro, M. Castro, J.-F. Mehaut, and A. Carrissimi, "Improving memory affinity of geophysics applications on NUMA platforms using Minas," in *Proc. Int. Conf. High Perform. Comput. Comput. Sci.*, 2010, pp. 279–292.

[18] J. Marathe, V. Thakkar, and F. Mueller, "Feedback-directed page placement for ccNUMA via Hardware-generated memory traces," *J. Parallel Distrib. Comput.*, vol. 70, pp. 1204–1219, 2010.

[19] W. J. Bolosky and M. L. Scott, "Evaluation of multiprocessor memory systems using Off-line optimal behavior," *J. Parallel Distrib. Comput.*, vol. 15, pp. 382–398, 1992.

[20] J. Marathe and F. Mueller, "Hardware profile-guided automatic page placement for ccNUMA systems," in *Proc. Symp. Principles Practice Parallel Programm.*, 2006, pp. 90–99.

[21] M. M. Tikir and J. K. Hollingsworth, "Hardware monitors for dynamic page migration," *J. Parallel Distrib. Comput.*, vol. 68, pp. 1186–1200, 2008.

[22] E. H. M. Cruz, M. Diener, M. A. Z. Alves, L. L. Pilla, and P. O. A. Navaux, "Optimizing memory locality using a Locality-aware page table," in *Proc. Int. Symp. Comput. Archit. High Perform. Comput.*, 2014, pp. 198–205.

[23] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis, "Autopin—Automated optimization of thread-to-core pinning on multicore systems," in *Proc. Workshop Programmability Issues Multi-Core Comput.*, 2008, pp. 1–12.

[24] P. Radojković, V. Cakarević, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Thread assignment of multithreaded network applications in multicore/multithreaded processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 12, pp. 2513–2525, Dec. 2013.

[25] M. Diener, E. H. M. Cruz, and P. O. A. Navaux, "Communication-based mapping using shared pages," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2013, pp. 700–711.

[26] M. A. Alves, "Increasing energy efficiency of processor caches via line usage predictors," Ph.D. dissertation, Federal Univ. of Rio Grande do Sul, Beazil, 2014.

[27] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "The McPAT framework for multicore and manycore architectures: simultaneously modeling power, area, and timing," *ACM Trans. Archit. Code Optim.*, vol. 10, pp. 1–29, 2013.

[28] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," NASA Ames Research Center, Tech. Rep. NAS-99-011, 1999.

[29] C. Luk, R. Cohn, R. Muth, and H. Patil, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. Conf. Programm. Lang. Des. Implementation*, 2005, pp. 190–200.

[30] E. H. M. Cruz, M. Diener, L. L. Pilla, and P. O. A. Navaux, "An efficient algorithm for communication-based task mapping," in *Proc. Int. Conf. Parallel, Distrib. Netw.-Based Process.*, 2015, pp. 207–214.

[31] F. Pellegrini, "Static mapping by dual recursive bipartitioning of process and architecture graphs," in *Proc. Scalable High-Perform. Comput. Conf.*, 1994, pp. 486–493.

[32] E. Jeannot, G. Mercier, and F. Tessier, "Process placement in multicore clusters: Algorithmic issues and practical techniques," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 4, pp. 993–1002, Apr. 2014.

[33] M. Woodacre, D. Robb, D. Roe, and K. Feind, "The SGI altix 3000 Global Shared-memory architecture," SGI, 2005.

[34] Intel, "Dual-core intel itanium processor 9000 and 9100 series," Intel, Tech. Rep. 314054-002, 2007.

[35] Intel, "Intel Xeon Processor 7500 Series," Intel, Tech. Rep. 323340-001, 2010.

[36] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek, "Intel quickpath Interconnect - architectural features supporting scalable system architectures," in *Proc. Symp. High Perform. Interconnects*, 2010, pp. 1–6.

[37] AMD, "AMD opteron 6300 series processor quick reference guide," AMD, 2012.

[38] L. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis," in *Proc. USENIX Annu. Tech. Conf.*, 1996, p. 23.

[39] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 72–81.

[40] Intel. (2012). Using KMP_AFFINITY to create OpenMP thread mapping to OS proc IDs. [Online]. Available: https://software.intel.com/en-us/articles/using-kmp-affinity-to-create-openmp-thread-mapping-to-os-proc-ids

Matthias Diener graduated in computer engineering at the TU Berlin and received the PhD degree in computer science at UFRGS. His research interests are in improving the memory access behavior of parallel applications that run on shared-memory architectures.

Eduardo H. M. Cruz received the master's degree at UFRGS, where he is currently working toward the PhD degree. His research focuses on improving the communication between threads on shared-memory architectures and to improve the locality of the memory accesses in architectures with nonuniform memory access.

Marco A. Z. Alves received the master's degree in 2009 and the PhD degree in 2014 from UFRGS. Currently, he is a postdoctoral researcher, focusing on Computer System Architecture.

Philippe O. A. Navaux graduated in electronic engineering from UFRGS in 1970, and received the masters degree in applied physics from UFRGS in 1973 and the PhD degree in computer science from INPG, France in 1979. He is a professor at UFRGS since 1973. He is the head of the Parallel and Distributed Processing Group at UFRGS and a consultant to various national and international funding agencies such as DoE (US), ANR (FR), CNPq, and CAPES (BR).

Anselm Busse graduated in computer engineering at Technische Universität Berlin and is currently working toward the PhD degree at this university. His research interests include process scheduling in operating systems, heterogeneous computing, and solid-state storage.

Hans-Ulrich Heiss received the PhD degree in computer science from the University of Karlsruhe. Since 2001, he is a full professor at TU Berlin. Currently, he is also a vice president for education at TU Berlin. In addition, he is a president of the German Council of University Faculties in engineering and informatics. His research interests include operating systems, distributed and parallel computing, and performance evaluation.