kMAF: Automatic Kernel-Level Management of Thread and Data Affinity

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

Eduardo H. M. Cruz
Informatics Institute, UFRGS
Porto Alegre, Brazil
ehmcruz@inf.ufrgs.br

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

Anselm Busse
Communication and Operating Systems Group, TU Berlin
Berlin, Germany
anselm.busse@tu-berlin.de

Hans-Ulrich Heiß
Communication and Operating Systems Group, TU Berlin
Berlin, Germany
hans-ulrich.heiss@tu-berlin.de

ABSTRACT

One of the main challenges for parallel architectures is the increasing complexity of the memory hierarchy, which consists of several levels of private and shared caches, as well as interconnections between separate memories in NUMA machines. To make full use of this hierarchy, it is necessary to improve the locality of memory accesses by reducing accesses to remote caches and memories, and using local ones instead. Two techniques can be used to increase the memory access locality: executing threads and processes that access shared data close to each other in the memory hierarchy (thread affinity), and placing the memory pages they access on the NUMA node they are executing on (data affinity). Most related work in this area focuses on either thread or data affinity, but not both, which limits the improvements. Other mechanisms require expensive operations, such as memory access traces or binary analysis, require changes to hardware or work only on specific parallel APIs.

In this paper, we introduce kMAF, a mechanism that automatically manages thread and data affinity on the kernel level. The memory access behavior of the running application is determined during its execution by analyzing its page faults. This information is used by kMAF to migrate threads and memory pages, such that the overall memory access locality is optimized. Extensive evaluation with 27 benchmarks from 4 benchmark suites shows substantial performance improvements, with results close to an oracle mechanism. Execution time was reduced by up to 35.7% (13.8% on average), while energy efficiency was improved by up to 34.6% (9.3% on average).

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.

ACM 978-1-4503-2809-8/14/08 ...$15.00.
http://dx.doi.org/10.1145/2628071.2628085.

Categories and Subject Descriptors

B.3.2 [**MEMORY STRUCTURES**]: Design Styles—*Cache memories, Virtual memory*; D.4.1 [**OPERATING SYSTEMS**]: Process Management—*scheduling*

Keywords

Thread affinity, Data affinity, NUMA, Cache hierarchies

1. INTRODUCTION

The memory hierarchy of modern parallel architectures presents challenges for thread scheduling and memory management of the operating system. When scheduling threads, their performance depends on which core each thread is executed, since the memory access performance varies depending on the core and the memory hierarchy [15]. These differences in the memory access latency between cores are due to shared cache levels, different bandwidths and latencies in the interconnections [38]. Regarding memory management, the memory hierarchy introduced non-uniform memory access (NUMA) due to multiple memory controllers in the system. Because of the NUMA characteristics, the performance of memory accesses also depends on which NUMA node each memory page is allocated, affecting overall performance and energy consumption [1]. For these reasons, it is important to consider the memory accesses of an application for thread scheduling and memory management [6].

Improving the locality by establishing an affinity of threads to cores and data to NUMA nodes has several benefits [35]. When threads that share data are executing on cores that share a cache, less cache misses are expected due to a reduction of cache line replications [10]. Moreover, when the cores executing threads that share data are nearby in the memory hierarchy, the latency to access the shared data is reduced. A lower overhead from cache coherence protocols can also be expected when shared data is not replicated [29]. Regarding data affinity, when the data accessed by the core is located on the local NUMA node, the access latency is lower compared to an access to a remote NUMA node [33].

Several previous proposals aim to optimize the affinity of threads and data considering the memory access behavior. Most approaches focus on either thread affinity [2, 13, 14] or data affinity [1, 12, 28, 34], but perform them only

separately. Some mechanisms rely on execution traces [14, 28], which cause a high overhead [3] and can not be used if the behavior of the application changes between executions. Mechanisms that focus on specific parallel programming paradigms [22, 30] are not a generic solution for data and thread affinities. Other approaches use indirect information about the memory access behavior [2], which can result in less accurate affinities.

In this paper, we propose a *kernel memory affinity framework (kMAF)*, which manages both thread and data affinity according to the memory access behavior of parallel applications. kMAF detects which data each thread accesses during the execution of the application by using the virtual memory implementation of the operating system. This information is used together with information about the hardware topology to calculate optimized affinities and migrate threads and memory pages.

kMAF presents several advantages compared to the current state-of-the-art. As the affinity management is performed on the kernel level, kMAF requires no changes to the application or its runtime libraries, and is therefore compatible with a large variety of parallelization models. Furthermore, it is hardware-independent and only requires that the architecture supports virtual memory with paging. As kMAF works completely during the execution of the parallel application, it needs no time-consuming analysis phase or any previous information about the behavior of the application.

We implemented kMAF in version 3.8 of the Linux kernel and evaluate it on a 4-node NUMA system using four sets of parallel benchmarks: The OpenMP implementation of the NAS Parallel Benchmarks (NPB) [21], PARSEC [4], which uses the Pthreads and OpenMP APIs, the MPI-based HPC Challenge Benchmark (HPCC) [26] and the hybrid MPI+OpenMP implementation of NPB [37]. Results show significant performance and energy efficiency improvements compared to the original thread and data affinity management of the Linux kernel.

The rest of the paper is organized as follows: In the next section, we introduce metrics to characterize the memory access behavior of parallel applications and evaluate their suitability for thread and data affinity. Section 3 introduces kMAF, our proposed mechanism. The evaluation methodology is shown in Section 4. Section 5 presents and analyzes experimental results. Related work is discussed in Section 6. Finally, Section 7 summarizes our conclusions.

2. BACKGROUND: AFFINITY IN SHARED MEMORY ARCHITECTURES

In this section, we begin with a brief outline of the benefits of managing the thread and data affinity of parallel applications. We then introduce a set of metrics that can be used to determine the potential for affinity management of a parallel application and use these metrics to analyze several sets of parallel benchmarks.

2.1 Benefits of Affinity Management

The management of thread and data affinity aims to improve the accesses to shared and private data in parallel applications.

Managing the thread affinity improves the efficiency of the interconnections, reducing inter-chip traffic that has a higher latency and lower bandwidth than intra-chip interconnections. It also reduces the number of cache misses of parallel applications. In read-only situations, executing threads on the same shared cache reduces data replication, thereby increasing the cache space available to the application [10]. In read-write or write-write situations, an optimized affinity also reduces cache line invalidations.

Managing data affinity improves the locality on NUMA machines by reducing the number of accesses to remote memory banks. As thread affinity, it improves the efficiency of the interconnections. This increases the memory bandwidth available in the system and reduces the average memory access latency.

It is important to note that thread affinity is a prerequisite for data affinity. Data affinity alone is not able to improve locality when more than one thread accesses the same pages, since the threads may be executing on cores of different NUMA nodes. In this situation, only the threads that are executing on the same node where the data is located would benefit from the locality. With the thread affinity, threads that share a lot of data would be executed on the same node, improving the effectiveness of the data affinity.

2.2 Quantifying Application Memory Access Behavior

To determine the importance of thread and data affinity for a parallel application, we introduce metrics to describe the memory accesses behavior of a parallel application.

2.2.1 Metrics for Thread Affinity

Throughout this paper, we refer to memory accesses to the same address by two different threads as *sharing*. As thread affinity management focuses mostly on caches, we look at sharing at the cache line granularity, which is 64 Bytes on most architectures. The first metric we introduce is based on the intuition that to benefit from thread affinity, it is necessary to have groups of threads that share the same data among each other, and not with other threads. Conversely, if all threads share the same amount of data in the same way among them, no improvements from thread affinity can be expected.

To formalize this intuition, consider a matrix M, in which each element $M[i][j]$ contains the amount of sharing between threads i and j. We can determine the potential for thread affinity of the application by analyzing the differences between the amount of sharing between the threads. For that purpose, we calculate the *heterogeneity* of the sharing behavior with Equation 1, where T is the number of threads. The higher the heterogeneity, the higher the potential for thread affinity, since there are higher differences between the amounts of sharing. If heterogeneity is low, the amount of sharing between the threads is similar, thus any thread affinity applied would result in similar performance for accesses to shared data.

$$Heterogeneity = \frac{\sum_{i=1}^{T}\sum_{j=1}^{T}\left(\frac{\sum_{k=1}^{T} M[i][k]}{T} - M[i][j]\right)^2}{T^2} \quad (1)$$

Apart from the heterogeneity, another important metric is the *amount* of sharing between the threads, which we calculate with Equation 2. When there is little sharing, thread affinity has less potential for improvements, even when the heterogeneity is high. Therefore, both the amount and het-

erogeneity of the sharing must be taken into account when analyzing the potential for thread affinity of a parallel application.

$$SharingAmount = \frac{\sum_{i=1}^{T}\sum_{j=1}^{T} M[i][j]}{T^2} \quad (2)$$

2.2.2 Metrics for Data Affinity

The potential for data affinity of a page is proportional to the amount of memory accesses from a single NUMA node. That is, if a page presents most of its accesses from the same node, it has more potential for data affinity than a page that is accessed from several nodes. We call the highest number of memory accesses to a page from a single NUMA node compared to the number of accesses from all nodes the *exclusivity* of the page. The higher the exclusivity of a page, the higher its potential for data affinity. The exclusivity for a page is calculated with Equation 3, where NA is a vector containing the number of memory accesses, each element $NA[i]$ is the number of memory accesses from node i, and N is the number of NUMA nodes.

$$PageExclusivity = \frac{max(NA)}{\sum_{i=1}^{N} NA[i]} \quad (3)$$

The exclusivity is minimal when a page has exactly the same amount of accesses from all nodes. In this case, the exclusivity is given by $1/N$. The exclusivity achieves its maximum when all accesses to the corresponding page originate from the same NUMA node. In this case, the exclusivity is 1.

It is also important to take the amount of memory accesses to a page into account. The higher the amount of accesses, the higher the potential for data affinity. Combining these two metrics, the pages that have a lot of memory accesses and high exclusivity are the ones with the highest potential for data affinity. It is important to note that the thread affinity influences both data affinity metrics, since they consider the NUMA node that generated the memory accesses, not the threads themselves. This underlines our previous statement that data affinity depends on thread affinity.

The exclusivity describes a single page. To calculate the average exclusivity for the whole application, we scale the exclusivity of each page with the number of memory accesses to it, and divide it by the total number of memory accesses. This operation is shown in Equation 4, where $PageExclusivity_i$ is the exclusivity of page i, $Accesses_i$ is the number of memory accesses to page i, and P is the number of pages. As for the page exclusivity, the minimum and maximum of the average exclusivity is $1/N$ and 1, respectively, where N is the number of NUMA nodes in the system.

$$AvgExclusivity = \frac{\sum_{i=1}^{P} PageExclusivity_i \cdot Accesses_i}{\sum_{i=1}^{P} Accesses_i} \quad (4)$$

Another requirement for data affinity is that the memory usage of the application must be significantly higher than the cache size of the system. If the memory that an application uses fits into the processor caches, only minor improvements from a data affinity policy can be expected, as the caches can filter most of the memory accesses.

2.3 Benchmark Examples

In this section, we analyze several benchmarks to determine their suitability for affinity management by using the metrics introduced in the previous section.

2.3.1 Methodology

We analyze the potential for thread and data affinities with applications from 4 benchmark suites, whose details are summarized in Table 1. We run the OpenMP version of the NAS Parallel Benchmarks (NPB) [21], in which all applications except DC were executed using input size *C*. For DC, input size *B* was chosen as it is the largest input size of this benchmark. The second benchmark suite is PARSEC [4], which contains applications that are based on the OpenMP and Pthreads APIs. It was executed using the *native* input size. The HPC Challenge benchmark (HPCC) [26] is a single application based on MPI that consists of 16 phases with different computational and memory access characteristics. The fourth benchmark suite is the Multi-zone version of the NAS Parallel Benchmarks [37], which are hybrid applications implemented using MPI and OpenMP.

To analyze these applications, we created a simulator for the Pin dynamic binary analysis tool [25] that monitors all memory accesses of the parallel application. We use this simulator to calculate the heterogeneity, amount of accesses to shared data and the exclusivity. For the thread affinity metrics, we consider memory accesses by 2 threads in the same cache line as sharing. For the data affinity, we will present results for page sizes of 4 KByte and 2 MByte. The benchmarks were executed with 64 threads on a simulated 4-node NUMA architecture. We assign the threads to the NUMA nodes in such away that the exclusivity is maximized.

2.3.2 Results

The average exclusivity of all pages is presented in Figure 1 for 4 KByte and 2 MByte page sizes. To visualize the exclusivity of individual pages, we round the exclusivity of each page to the nearest 10% and group pages with the same rounded exclusivity together. Figure 2 depicts this visualization for 4 KByte pages. Figure 3 shows the values for heterogeneity and amount of sharing.

Table 1: Overview of the benchmarks used in the evaluation, showing parallelization API, benchmark names, input size and average memory usage per application.

Benchmark suite	Parallel API	Benchmark names	Input size	Memory usage
NAS-OMP v3.3.1	OpenMP	BT, CG, DC, EP, FT, IS, LU, MG, SP, UA	*C* DC:*B*	6.1 GByte
PARSEC v2.1	Pthreads, OpenMP	Blackscholes, Bodytrack, Facesim, Ferret, Freqmine, Raytrace, Swaptions, Fluidanimate, Vips, X264, Canneal, Dedup, Streamcluster	*native*	4.8 GByte
HPCC v1.4.3	MPI	HPCC (1 application with 16 phases)	4000^2 matrix	19.3 GByte
NAS-MZ v3.3.1	MPI, OpenMP	BT-MZ, LU-MZ, SP-MZ	*C*	5.8 GByte

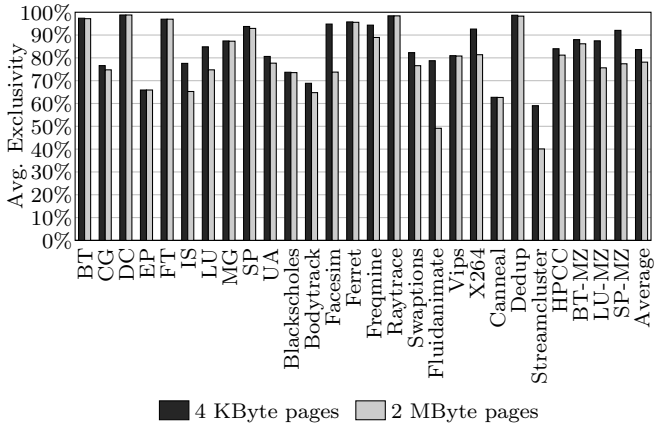


Figure 1: Average exclusivity for different page sizes.

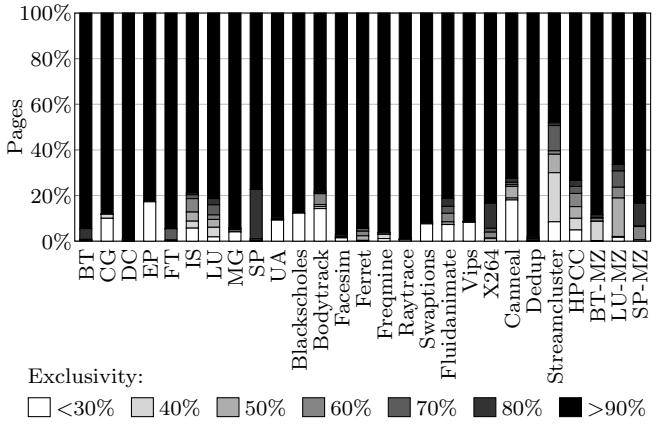


Figure 2: Page exclusivity for 4 KByte pages. The rounded exclusivity is shown in gray scale.

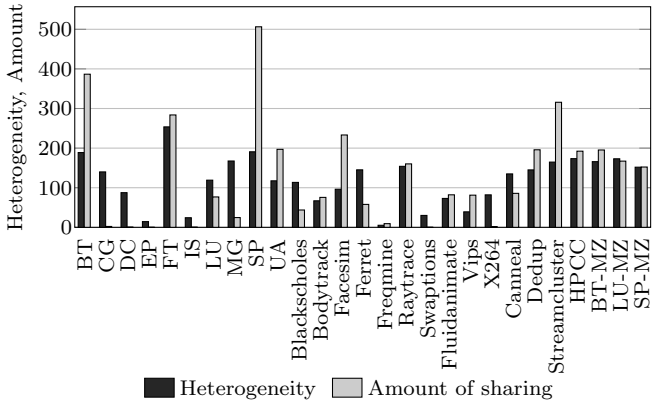


Figure 3: Sharing patterns.

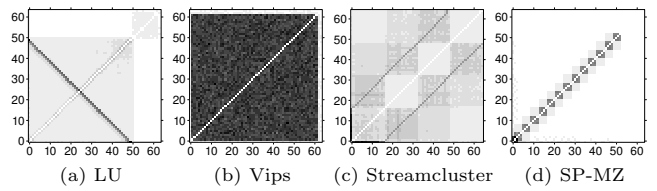


Figure 4: Example affinity matrices. Axes represent thread IDs. Cells show the amount of sharing for each pair of threads or processes. Darker cells indicate more sharing.

In most applications, the exclusivity is high, showing the importance of data affinity. Even in Streamcluster, which presents the lowest value, approximately 60% of its pages have a high exclusivity. On the other hand, the metrics for thread affinity (heterogeneity and amount of sharing), are not high for all applications, such as in Freqmine and EP. In these applications, although there is not much sharing between the threads, data affinity is important because each thread still needs to access its private data.

Some applications, such as SP and Dedup, have high values for exclusivity, heterogeneity and amount of sharing, and therefore can benefit from both thread and data affinity. CG and DC have a high heterogeneity but only share little data. EP and Swaptions have a very low total memory usage (less than 65 MByte), such that few improvements from data affinity are expected, since most of the memory they use fits into the caches. It is important to note that even when increasing the page size from 4 KByte to 2 MByte, the exclusivity only decreases slightly, from 83.6% to 78.3% on average for all benchmarks.

Figure 4 shows the sharing patterns of four benchmarks. Darker cells indicate more sharing between threads or processes. In several benchmarks, such as BT, MG and SP, most sharing happens between neighboring threads, such as threads 0 and 1. LU shows data sharing between more distant threads, with almost no sharing between threads 52–63. In Vips, the amount of sharing between all threads is almost the same. It therefore has a low heterogeneity, as indicated by Figure 3. Streamcluster shows a pipeline sharing pattern, where groups of threads of the same pipeline stage access the same data. HPCC has a complex pattern with sharing between neighboring and more distant threads. Additionally, nearly every of the 16 phases has a different sharing pattern.

2.3.3 *Summary of Benchmark Behavior*

Summarizing our analysis of the benchmarks, we can affirm that a large majority of them are suitable for data affinity. Less benchmarks are suitable for thread affinity. However, for the reasons mentioned previously, it is necessary to manage the thread affinity to fully benefit from the data affinity. Another important result is that when increasing the page size from 4 KByte to 2 MByte, the exclusivity of the memory accesses of most applications decreases only very slightly. This indicates that data affinity is important even when considering large pages.

3. KMAF: AUTOMATIC AFFINITY MANAGEMENT IN THE KERNEL

Our proposal, the *Kernel Memory Affinity Framework (kMAF)*, uses the virtual memory implementation of the operating system to characterize the memory access behavior of parallel applications during their execution and uses this information to optimize the affinity of the running application. kMAF requires no changes to the hardware, and is compatible with all architectures that use virtual memory with paging.

An overview of kMAF is shown in Figure 5. It consists of the following four parts, denoted A–D in the figure:
A. Determine the memory access behavior. Information about the parallel applications' memory access behavior is collected at the OS level through its page faults.

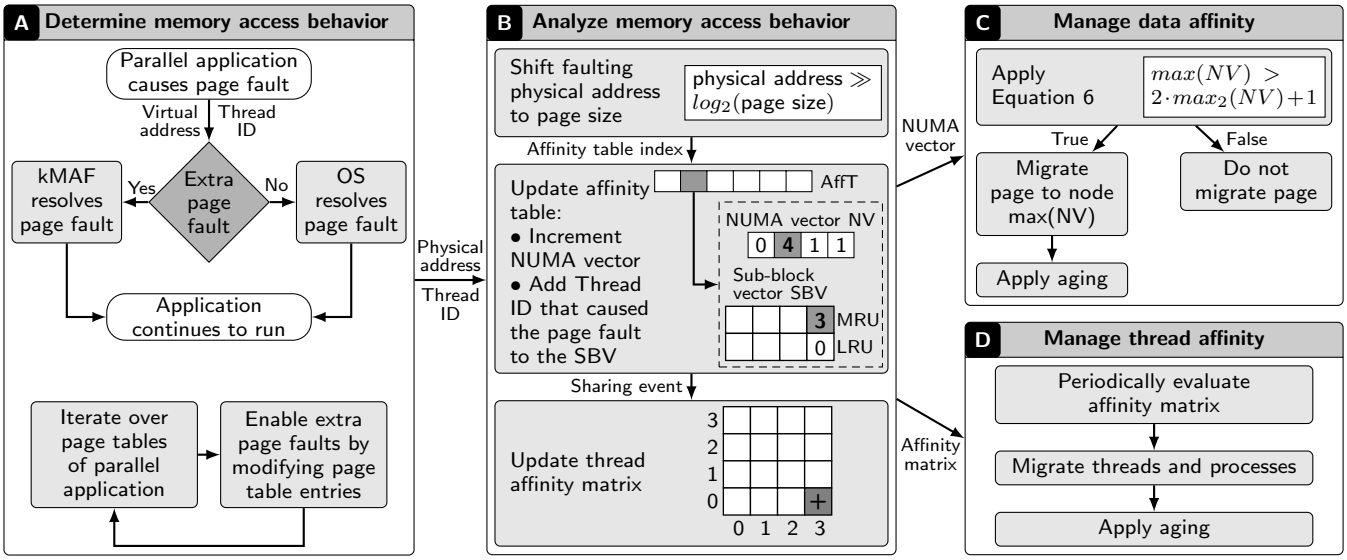


Figure 5: Overview of kMAF. In the example, a parallel application consisting of four threads is executing. The system contains 4 NUMA nodes. Each page is split into 4 sub-blocks, with 2 positions for threads (MRU and LRU). Thread 3, executing on NUMA node 1, causes a page fault. kMAF increments the NUMA vector in position 1, and adds thread 3 to the sub-block vector in the MRU position. Since the sub-block had been accessed previously by thread 0, it is moved to the LRU position, and a sharing event is stored in the affinity matrix by incrementing it in cell (0,3). The NUMA vector is evaluated by Equation 6, and the page gets migrated to node 1, since the condition is fulfilled.

B. Analyze the memory access behavior. The information about memory accesses is stored in a per-page list, together with the NUMA nodes that accessed the page and the threads that accessed sub-blocks of the page.
C. Manage data affinity. The per-node number of accesses to memory pages is analyzed to check if they should be migrated to a different NUMA node.
D. Manage thread affinity. kMAF periodically applies a mapping algorithm to determine if threads need to be migrated.

3.1 Determine Memory Access Behavior

Gathering information about memory accesses at the operating system level presents a challenge, as the accesses are usually performed directly by the hardware. Previous research [11, 13] has shown that memory access information can be gathered by analyzing the page faults of parallel applications. We adapt the idea of these mechanisms for kMAF to determine the memory access behavior. It is based on two fundamental concepts, which are described in this section.

3.1.1 Sampling Memory Accesses via Page Faults

Whenever an application accesses a memory address and its corresponding page is not set as present in the page table, the OS is notified about the faulting virtual address, as well as the thread ID that caused the fault. By tracking page faults that happen in the same memory area, it is possible to detect which threads are accessing shared data. Threads belonging to the same process use the same page table and therefore share the same virtual and physical address space. If threads belong to different processes, they do not share a page table. Shared memory regions between threads from different processes have the same physical memory address, but not necessarily the same virtual address. In order to support the detection of sharing in these multi-process applications, kMAF uses the physical address for detection. Since the full address is available during the page fault, it is possible to use different memory area sizes, such as the cache line or page size, for detection. This also makes the detection independent from the page size of the hardware.

3.1.2 Increasing Accuracy via Extra Page Faults

It is important to note that only one page fault per page happens during normal operation, which would limit the accuracy of the detection mechanism. This limitation can be overcome by enabling multiple page faults per page. These extra page faults are injected by kMAF by periodically iterating over the page tables of the parallel application and clearing the page present bit of pages that can be shared between threads. As these extra page faults do not represent missing information in the page table, they can be resolved quickly by kMAF itself without the normal kernel routines, thus minimizing their overhead.

In processes that contain only one thread, kMAF only enables extra page faults in pages that are shared with other processes. In multi-threaded processes, extra page faults are enabled for all data segments, since the operating system does not keep track of which threads access each page.

3.2 Analyze Memory Access Behavior

We store information about the memory access pattern in an *Affinity Table (AffT)*. Each table entry contains information about one physical page frame. To locate the entry of a page in the affinity table, we bit shift the physical address to the right, removing the offset bits inside a page, as shown in Equation 5.

$$AffTIndex = PhysAddr \gg log_2(PageSize) \quad (5)$$

In each affinity table entry, kMAF stores the number of accesses from each NUMA node to each page in the *NUMA*

vector (NV). The number of counters in the NUMA vector is equal to the number of NUMA nodes in the system. The number of bits used to store each counter can be configured. A high number of bits provides more accuracy, but requires more memory to store the counter.

To manage thread affinity, we store the IDs of the threads that access each page in the *sub-block vector (SBV)*. As we need a finer granularity than the page size to determine thread affinity, each page is divided into several sub-blocks of equal size. To support this feature, each entry of the affinity table contains several fields that store the threads that access each sub-block of the page. For each sub-block, the number of IDs of threads that access it can be configured. Tracking more threads can lead to a higher accuracy, but requires more memory and could lead to false positives. False positives may occur because threads that have not accessed the sub-block for a long time should not be considered a sharer of the sub-block.

The thread ID that generated the page fault is stored in the MRU position in the SBV entry that corresponds to the sub-block of the memory address. In case the sub-block has been accessed before by other threads, these other threads are moved towards the LRU position. Furthermore, we record a sharing event by incrementing the affinity matrix in the cells that correspond to all pairs of threads in the SBV entry.

3.3 Manage Data Affinity

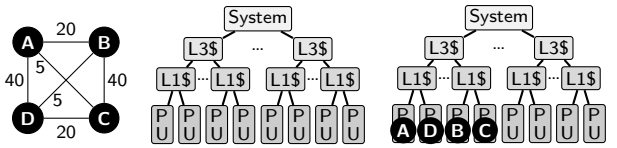
To manage the data affinity, on every page fault kMAF evaluates the NUMA vector of the page that caused the fault to determine the exclusivity of the page. We calculate the exclusivity by looking at the number of accesses from the NUMA nodes. If the highest number of accesses from a node is more than double the second highest number, the page will be migrated to the node with the highest number of accesses, if it is not already located on that node. Equation 6 formalizes this behavior, where $max(NV)$ is the highest value of the NUMA vector, and $max_2(NV)$ is the second highest value of the NUMA vector.

$$Migrate = \begin{cases} true & \text{if } max(NV) > 2 \cdot max_2(NV) + 1 \\ false & \text{otherwise} \end{cases} \tag{6}$$

The idea behind this equation is to prevent early migration of pages, during application initialization, but enable quick migration as soon as a pattern is established. We delay any page migration until we set the thread affinity for the first time, to avoid unnecessary page migrations due to page sharing between the threads, and to benefit from a higher exclusivity of the pages when threads that access the same page execute on the same NUMA node. We also perform aging in the NUMA vector to be able to better react to changes in the memory access behavior. Every time a page is migrated, we divide all counters of the NUMA vector by 2 to reduce the impact of old values on the data affinity.

3.4 Manage Thread Affinity

To optimize the mapping of processes and threads to processing units (PUs), kMAF periodically evaluates the thread affinity matrix. To be able to react quickly to changes in the memory access behavior, we selected a quick graph partitioning algorithm that generates the mapping and execute it with a period of 100 ms. Furthermore, we apply an aging mechanism to the affinity matrix to reduce the influence of old values in the matrix.



(a) Input: Affinity graph. (b) Input: Hierarchy graph of the machine. (c) Output: Mapping of threads to PUs.

Figure 6: Inputs and output of the mapping algorithm.

3.4.1 Calculating the Thread Mapping

The information provided by the sharing detection is used to calculate an optimized mapping from processes and threads to PUs during the execution of the parallel application. The mapping problem is NP-hard [5], therefore it is necessary to use efficient heuristic algorithms to calculate the mapping. Online mapping requires algorithms with a short execution time, since it directly impacts the overhead on the executing application.

The mapping problem is modeled with two undirected graphs, an affinity graph and a hierarchy graph. In the affinity graph, vertices represent threads and edges represent the amount of affinity between them. Figure 6a shows an example affinity graph for four threads A–D. In the hierarchy graph, vertices represent levels of the memory hierarchy and PUs, while edges represent the interconnections between them. In the hierarchy graph, we do not consider levels whose arity is 1. For instance, if there is only one L3 cache for each NUMA node, there is no need to include the NUMA node level in the hierarchy graph when calculating the thread affinity. Figure 6b depicts a simplified hierarchy graph for the hardware architecture that will be used in the evaluation. The affinity graph is obtained from the affinity matrix during runtime, while the hierarchy graph is generated from the hardware information provided by the kernel.

To calculate the mapping, we use the dual recursive bipartitioning algorithm of the Scotch mapping library [31], version 6.0. We selected this algorithm because it has a short execution time while providing good results [32], and is therefore suitable for online mapping. The algorithm receives the affinity and hierarchy graphs as input, and outputs the PU for each thread such that the total cost of memory accesses is minimized, as shown in Figure 6c. This information is used to migrate the threads to their assigned PUs.

3.4.2 Aging Mechanism

After calculating the mapping, we apply an aging technique in the affinity matrix, multiplying its values by a factor of 0.75, to reduce the influence of old values in the matrix and make it easier to detect changes in the memory access behavior during execution. We also evaluated other factors between 0.6 and 0.95, but the results are not very sensitive to this value. We chose the value 0.75 because the operation can be performed efficiently with a subtraction and a bit shift operation: $M_{new}[i][j] = M_{old}[i][j] - (M_{old}[i][j] \gg 2)$, where i and j are the indices of the affinity matrices M_{old} and M_{new}.

3.5 Overhead of kMAF

kMAF introduces storage and runtime overheads, which will be discussed in this section.

3.5.1 Storage Overhead

The storage overhead is the amount of memory required to store the affinity table and thread affinity matrix.

Affinity Table. Equation 7 describes the size of each entry in the affinity table, $AffTentry_{size}$. In the equation, $NVcounter_{size}$ is the number of Bytes used to store each counter of the NUMA vector, N is the number of NUMA nodes, $Sub\text{-}block_{size}$ is the granularity chosen to detect thread affinity, $ThreadID_{size}$ is the number of bytes required to store a thread ID in the sub-block vector, and $\#Sharers$ is the number of tracked threads per sub-block. Equation 8 calculates the size of the affinity table, where $\#AffTentry$ is the number of entries of the affinity table.

$$AffTentry_{size} = NVcounter_{size} \cdot N + \frac{Page_{size}}{Sub\text{-}block_{size}} \cdot ThreadID_{size} \cdot \#Sharers \quad (7)$$

$$MemUsageAffT = \#AffTentry \cdot AffTentry_{size} \quad (8)$$

To calculate the storage overhead, we use the following configuration: $NVcounter_{size}$: 2 Bytes, $Sub\text{-}block_{size}$: 1 KByte, $ThreadID_{size}$: 2 Bytes and $\#Sharers$: 2. The system has a $Page_{size}$ of 4 KByte and 4 nodes. The same configuration will be used in the experiments. In this scenario, each entry of the affinity table requires 20 Bytes and the affinity table overhead is 0.49% of the total memory in the system.

Thread Affinity Matrix. The memory usage of the thread affinity matrix is given by Equation 9, where T is the number of threads of the parallel application and $TAffMentry_{size}$ is the size of each element of the matrix in bytes. For an element size of 4 Bytes and an application with 64 threads, the memory usage of the thread affinity matrix is 16 KByte.

$$MemUsageTAffM = T^2 \cdot TAffMentry_{size} \quad (9)$$

3.5.2 Runtime Overhead

The runtime overhead consists of the time required to introduce extra page faults, resolve these faults, and calculate thread and data affinity, plus the migrations. The complexity to introduce extra page faults increases linearly with the number of processes of the parallel application, as we need to iterate over each page table. Resolving the extra page faults from kMAF has a constant time complexity. The complexity of calculating the thread affinity depends on the algorithm adopted. The algorithm we adopted has a complexity of $O(T^3)$, where T is the number of threads of the parallel application [18]. For data affinity, the time complexity is $O(N)$, where N is the number of NUMA nodes. The overhead of the migrations depends on the actual implementation in the kernel. In Section 5.4, we will evaluate the runtime overhead on a running application.

3.6 Summary of kMAF

An example of the operation of kMAF is shown in the caption of Figure 5. kMAF is independent from the parallelization model, as the memory access behavior is determined directly from the memory accesses themselves. For the same reason, it also supports hybrid applications that use several models. kMAF works during the execution of the parallel application, and requires no modification to the application, its runtime system or expensive operations such as tracing. Methods that rely on tracing are not able to handle applications in which the access behavior changes between or during executions. Furthermore, changing the input data or the number of threads can also lead to different behaviors.

kMAF is also highly OS and hardware independent. It can be applied to all platforms that use virtual memory with paging, which covers the majority of the current computer architectures. No architecture specific information, such as a special hardware counter, is necessary. As the detection is completely dynamic, kMAF can be combined with other techniques that alter the behavior of a parallel application, such as modifying the number of processes and threads during execution [23] and migrating MPI processes between cluster nodes [9].

4. METHODOLOGY

We evaluated kMAF on a 4-node NUMA machine, where each node consists of an 8-core Intel Xeon X7550 processor that supports Simultaneous Multithreading (SMT). Each core has private L1/L2 caches, while the L3 cache is shared between all the cores in the processor. The machine parameters are summarized in Table 2.

We experimented with the same 27 benchmarks and input sizes that were described in Section 2.3.1. As the machine can run 64 threads concurrently, most benchmarks were executed with this number of threads or processes. In benchmarks where this is not possible, such as X264 and Streamcluster, we executed with a number as close as possible to 64. For the hybrid NAS-MZ benchmarks, several configurations of processes and threads were evaluated: 1 process with 64 threads, 2 processes with 32 threads each and 4 processes with 16 threads each. As the results were very similar, we show only the values for the 4 processes with 16 threads configuration. All benchmarks were compiled with gcc, version 4.6.3. For the benchmarks that use MPI, we used the Open MPI framework [16], version 1.5.1, which uses KNEM [17] for communication in shared memory.

Five different mapping mechanisms were compared:

Operating System (OS). The OS is the baseline for the experiments. We use an unmodified Linux kernel, version 3.8. We evaluated the standard first-touch policy and the AutoNUMA approach [11]. However, results were very close to each other (less than 2% difference), so we only show the results using the first-touch policy.

Oracle. For the oracle mechanism, we collected memory access traces as described in Section 2.3.1, and used information from the traces to perform an optimized thread and data mapping for the applications.

Table 2: Configuration of the evaluation system and kMAF.

	Parameter	Value
System	Processors	4x Intel Xeon X7550, 8 cores, 2-SMT
	Caches	32 KByte L1, 256 KByte L2, 18 MByte L3
	Memory	128 GByte DDR3-1333, 4 KByte page size
kMAF	NUMA vector	4x 1 Byte per page
	Sub-block vector	4x 2 entries per page (1 KByte size)
	Thread ID size	16 Bits each

Compact. The compact mapping, similar to existing implementations in some OpenMP frameworks [20], performs a round-robin scheduling of threads to cores that takes into account the memory hierarchy.
Random. For the random mapping, we generated random mappings of threads to cores and memory pages to NUMA nodes, one for each execution.
kMAF. We implemented kMAF in version 3.8 of the Linux kernel and use it with the configuration shown in Table 2.

All experiments were executed 20 times. For each experiment, we show the average values, as well as the confidence interval for a confidence level of 95% in a Student's t-distribution. All results are normalized to the baseline, the OS. We measured execution time, L3 cache misses and traffic on the QuickPath Interconnect (QPI) with the help of the Intel performance counter monitor (PCM) [19].

5. RESULTS

This section presents and discusses the results of kMAF compared to other affinity mechanisms, following the methodology shown in Section 4. We present four sets of results: performance, energy consumption, performance using only thread or data affinity, and the runtime overhead of kMAF.

5.1 Performance

The results of execution time, L3 cache misses and QPI traffic can be found in Figures 7, 8 and 9, respectively. The reduction of the cache misses, with an average of 11.1% in kMAF, are due to the more efficient thread affinity, since it reduces the cache line replication. Data affinity alone does not have a significant influence on cache misses. The reduction of the QPI traffic, with an average of 19.0%, happened due to the optimized thread and data affinity, as the data is located more often in the local memory banks. Both these metrics were responsible for the reduction of the execution time, with an average of 13.8% for kMAF.

To illustrate how thread affinity also affects data affinity, consider Streamcluster. In Streamcluster, the exclusivity is low but the heterogeneity and amount of sharing is high. However, the reduction of QPI traffic is higher than the reduction of cache misses. The reason is that the better fitting thread affinity results in a placement of threads that share data on the same NUMA node, thus reducing QPI traffic. Cache misses were not reduced to the same degree. Therefore, although the indicators of Streamcluster show a higher potential for thread affinity, we are able to observe this by looking at QPI traffic, not at cache misses.

For applications that are more influenced by data affinity than thread affinity, the performance is similar regardless of the thread affinity applied. For instance, Vips has a high exclusivity and low heterogeneity and amount of sharing, which makes its reduction of QPI traffic larger than the reduction of cache misses. However, these indicators do not always correlate. One example is DC, whose indicators are similar to the ones of Vips, but the results show a higher reduction of cache misses than QPI traffic.

Most applications are more sensitive to the data affinity than the thread affinity, which can be observed in the results by the fact that the QPI traffic presented a higher reduction than cache misses. This happens because, even if an application does not share much data among its threads, each thread will still need to access its own private data, which can only be improved through data affinity. It is important to note that this does not mean that data affinity is more important than thread affinity, because, as previously stated, the effectiveness of data affinity depends on thread affinity.

For each benchmark, several affinity policies were tested. On average, execution time was reduced by 13.8%, 15.3%, 3.2% and 1.2% compared to the OS by kMAF, Oracle, Compact and Random, respectively. We can observe that kMAF presented performance improvements close to the Oracle, which demonstrates that kMAF handled both thread and data affinities correctly. By analyzing the Compact and Random affinities, we note that simple affinity strategies are not sufficient to handle a variety of applications, since they have very different characteristics. Also, this shows that our performance improvements are due to a better affinity, and not due to unnecessary migrations introduced by the operating system.

5.2 Energy Consumption

Apart from performance, energy improvements represent an important goal of affinity management. Optimizing affinity can reduce the energy consumption through a more efficient use of the computing resources, such as the caches and interconnections. Furthermore, reducing execution time by itself also saves static energy. We measure the system energy consumption during the execution of each application with the help of the Baseboard Management Controller (BMC), which exposes the energy consumption of the system through IPMI. We compare kMAF to the same affinity mechanisms that were used in the performance experiments.

Figure 10 contains the energy consumption results normalized to the results of the baseline (OS). As expected, the applications that benefit more from affinity management show higher energy savings. As in the performance results, SP had the highest reduction of energy consumption, of 34.6%. On average, energy consumption was reduced by 9.3% by kMAF and by 11.1% by the Oracle. As before, Compact and Random only reduced energy consumption slightly compared to the baseline (4.1% and 2.9%, respectively).

5.3 Managing Thread and Data Affinity Separately

In Section 2.1, we mentioned that setting thread affinity is a requirement for data affinity. To evaluate the influence of thread and data affinity, we executed kMAF only with the thread affinity and data affinity part, and let the OS handle the respective other affinity. We use the OS thread and data affinity as the baseline, as before.

Figure 11 shows the execution time for the three configurations, normalized to the baseline. For most benchmarks, the improvements from the thread affinity only are higher than for the data affinity only. On average, data affinity reduced execution time by 2.0%, thread affinity by 6.4%. It is important to note that the improvements from managing thread and data affinity jointly, as shown in Section 5.1, are higher than the sum of improvements when managing the affinities separately. This shows that integrated mechanisms are necessary for optimal results.

5.4 Runtime Overhead

Since kMAF operates during the execution of the parallel application, it causes an overhead, as discussed in Section 3.5.2. The overhead can be attributed to the following

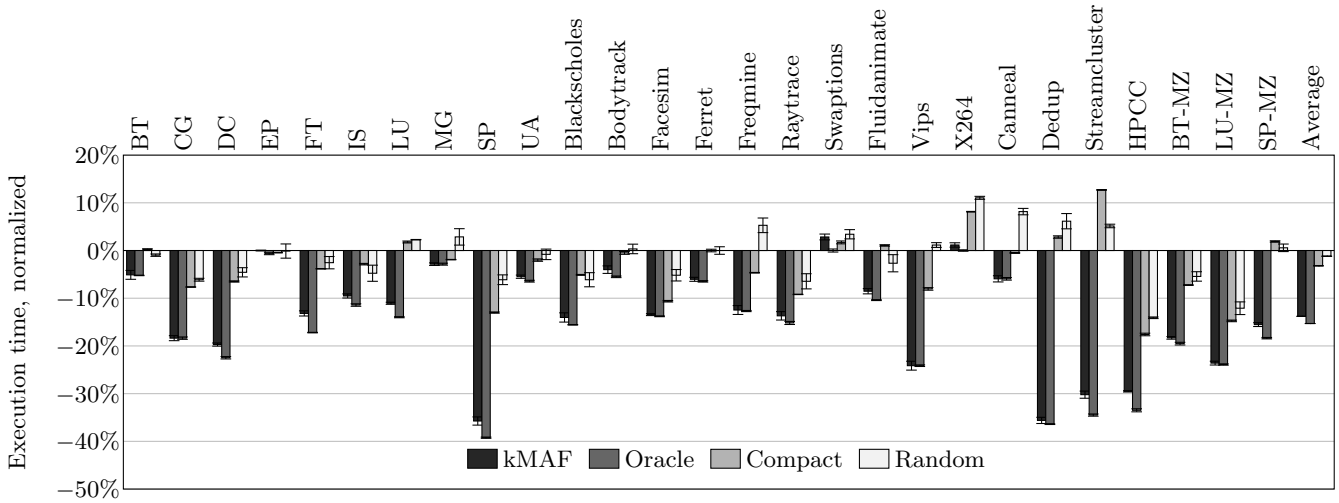


Figure 7: Execution time, normalized to the OS.

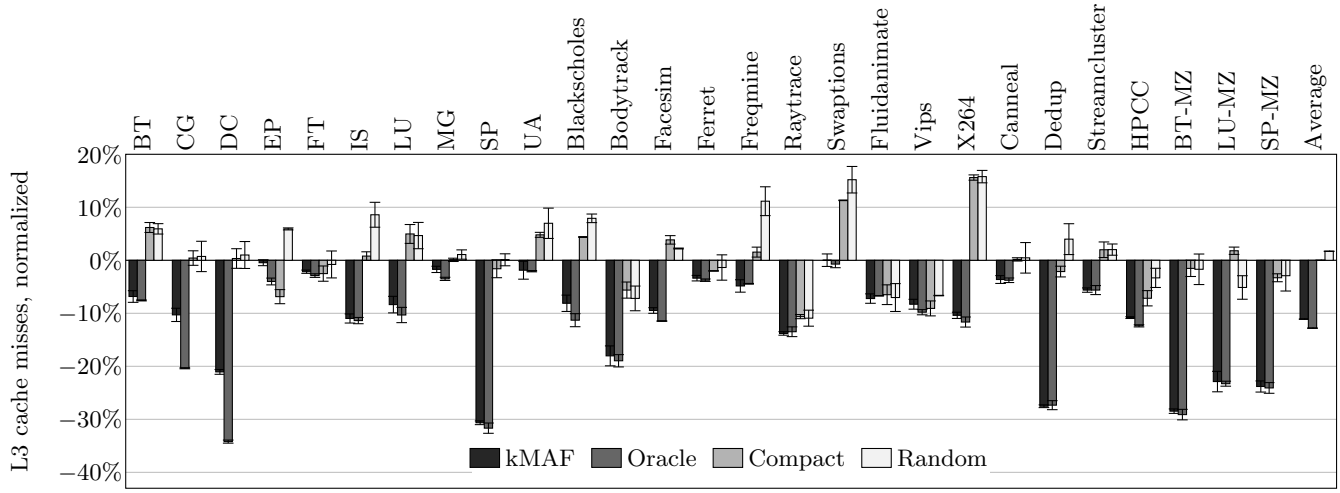


Figure 8: L3 cache misses, normalized to the OS.

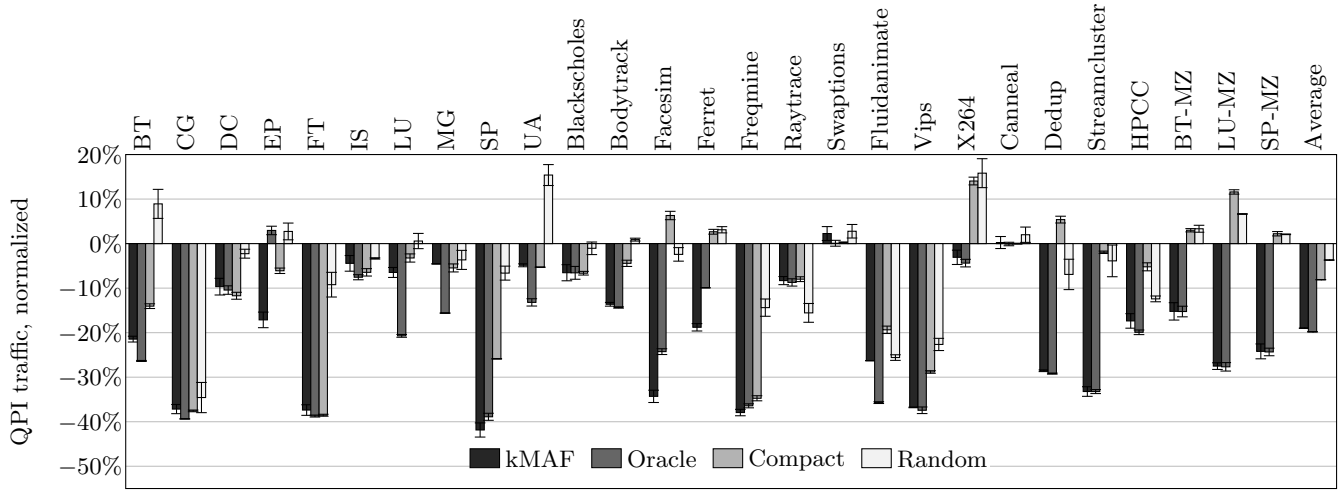


Figure 9: QPI traffic, normalized to the OS.

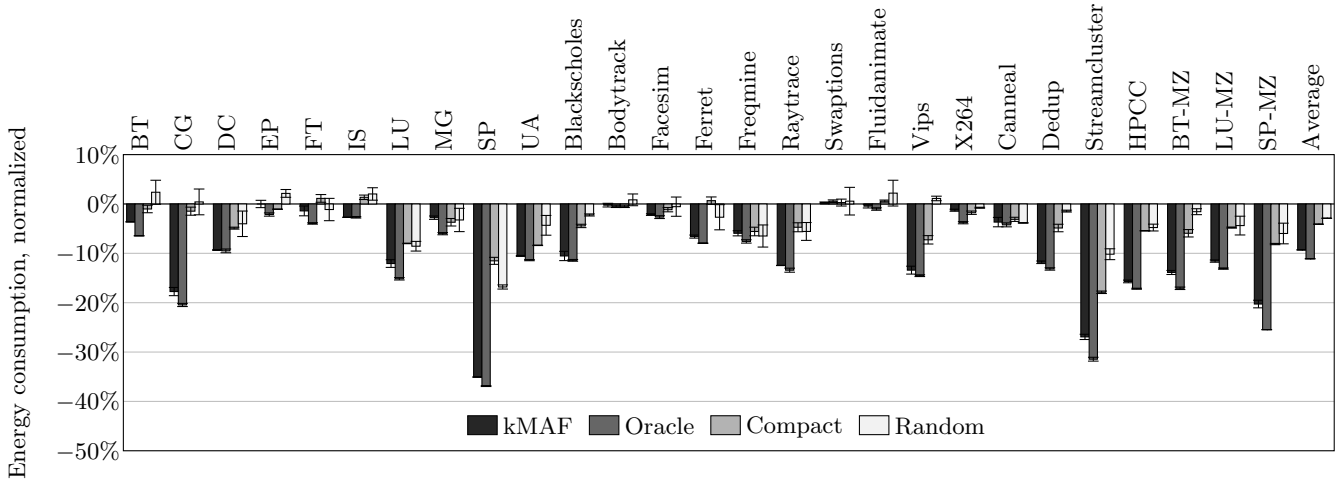


Figure 10: Energy consumption, normalized to the OS.

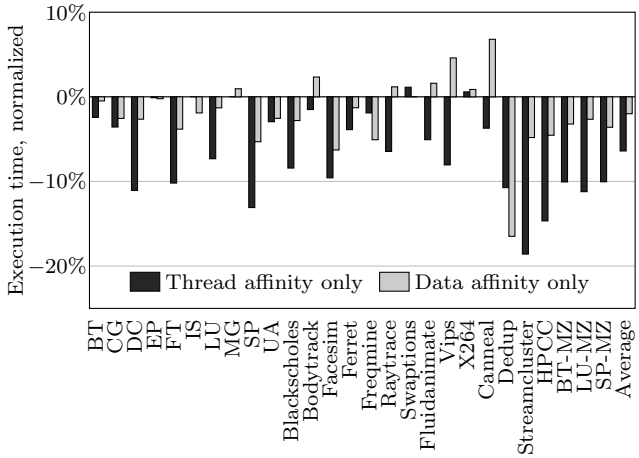


Figure 11: Application execution time when running kMAF with only thread or data affinity, normalized to the OS.

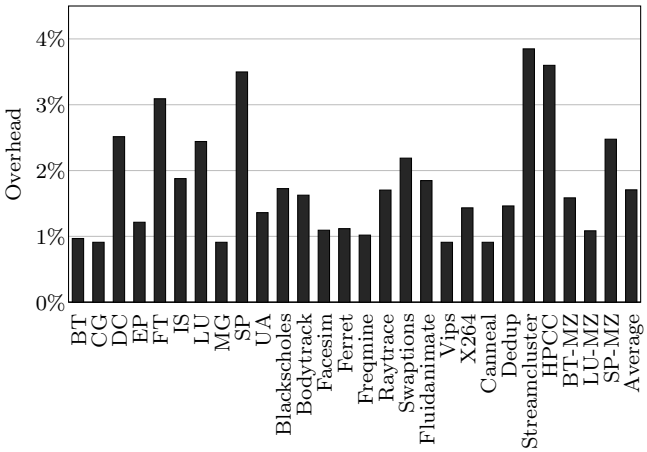


Figure 12: Total runtime overhead of kMAF as the percentage of the execution time of each application.

reasons: Extra page faults cause additional interrupts to the operating system, accesses to the affinity table and matrix, migration of pages between NUMA nodes, calculation of the thread mapping and the migration of threads. To approximate the overhead for the application, we measured the time spent in the kernel routines for each source of overhead, such as the time spent in the page fault handler for the extra page faults. Figure 12 shows the overhead for each application, as the percentage of each application's execution time. For all benchmarks, the overhead is less than 4%, with an average of 1.8%.

6. RELATED WORK

Several related mechanisms that focus on thread and process affinity considering the memory accesses have been proposed. In parallel applications that use explicit functions to share data, such as MPI communication functions, most previous proposals make use of information gathered from MPI primitives to generate a communication pattern. Frameworks such as the MPI Parallel Environment (MPE) [8] and eztrace [36] can be used to trace MPI messages between processes. Process affinity of MPI based applications is also evaluated in [7], but with the assumption that the affinities are previously known. In [30], the affinity can be either provided by the programmer, or be automatically detected by monitoring the MPI messages, as in [9]. A similar technique is used in [22]. These mechanisms utilize MPI primitives and are therefore not a generic solution for process and thread affinities. For multithreaded applications based on shared-memory frameworks, such as OpenMP and Pthreads, most traditional mapping mechanisms use memory access traces [3, 14] to characterize and map the application's threads. Tracing has a high overhead and can not be used if the application's behavior changes between executions.

Traditional data affinity policies, such as *first-touch*, *next-touch* [24] and *interleave* [12], have been used by operating systems to allocate memory on NUMA machines. In the case of first-touch, which is the default placement policy of Linux, a memory page is allocated in the NUMA node of the thread that causes its first page fault, and the page is never migrated. In the next-touch policy, pages are migrated to the node that performs the next access to them, leading to many migrations in pages that are accessed from different nodes. Recent versions of the Linux kernel can optionally

balance the memory pages between NUMA nodes (AutoNUMA [11]). Both the interleaving policy and the NUMA balancing did not have a measurable influence on the results for the benchmarks we executed (less than 2% difference compared to first-touch).

Different approaches have been proposed to overcome these issues. For instance, Awasthi et al. [1] use information from the memory controller, such as the queuing delays and row-buffer hit rates, to guide data affinity policies. However, this mechanism does not consider thread affinity. Similarly, Marathe et al. [28] present a page placement scheme for NUMA platforms. Their framework captures information from hardware counters to feed an affinity decision logic. However, their work requires the identification of the stable execution phase of the parallel application through traces.

Tikir and Hollingsworth [34] use UltraSPARC III hardware monitors to guide data affinity, but do not set thread affinity. They propose adding an address translation counter to the TLB, and gather statistics from cache misses and TLB misses. Their proposal is limited to architectures with software-managed TLBs, while ours works in any architecture that supports virtual memory. Similarly, Marathe and Mueller [27] use the Power Management Unit (PMU) of the Itanium-2 architecture to perform data mapping. Their profiling mechanism imposes a high overhead, as it requires traps to the operating system on every high latency memory load operation or TLB miss. Hence, they enable the profiling mechanism just during the beginning of each application, losing the opportunity to handle changes during the execution of the application. Carrefour [12] uses instruction based sampling (IBS), available on AMD architectures, to collect statistics about page accesses during runtime, and uses them to improve data affinity. Data affinity alone is not able to improve locality when more than one thread accesses the same pages, since the threads may be executed in cores of different NUMA nodes.

Azimi et al. [2] set the affinity of threads based on information from the hardware counters of Power5 processors that sample the memory addresses of accesses resolved by remote caches. Accesses solved by local caches are not considered, generating an incomplete thread affinity pattern. Our previous work, SPCD [13], uses the page faults of a parallel application to set thread affinity. Only thread affinity was performed in these mechanisms, which does not improve the locality of memory accesses in NUMA architectures.

7. CONCLUSIONS

Parallel architectures with complex memory access characteristics represent the state-of-the-art. To fully benefit from these architectures, it is important to analyze the memory access characteristics of parallel applications and use this information to optimize the locality of memory accesses. This can be done in two complementing ways: by running threads that access shared data close to each other in the memory hierarchy (thread affinity) and to place memory pages on the NUMA node that accesses them the most (data affinity).

In this paper, we presented metrics and a methodology to analyze the memory access behavior of parallel applications to determine their potential for thread and data affinity. We introduced kMAF, which is a framework that automatically manages affinity on the operating system level. It requires no hardware changes and is compatible with all architectures that use virtual memory with paging. Furthermore, it requires no previous information about the applications' behavior, and no changes to the applications themselves or their runtime libraries.

Experiments with a wide range of parallel applications with different memory access characteristics showed that kMAF was able to improve performance substantially, up to 35.7% (13.8% on average). Energy consumption was also reduced, by up to 34.6% (9.3% on average). Results were close to an oracle mechanism, and substantially better than simpler mechanisms that do not take application behavior into account. We also show that combining thread and data affinity can lead to higher improvements than performing them separately.

For the future, we intend to expand kMAF to the system as a whole, without focusing on a single running application.

Acknowledgment

The authors thank Laércio L. Pilla, Francis B. Moreira, Emmanuell D. Carreño and the anonymous reviewers for their comments. This work was partially supported by CNPq, FAPERGS and CAPES.

8. 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 *Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[2] R. Azimi, D. K. Tam, L. Soares, and M. Stumm, "Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, Apr. 2009.

[3] N. Barrow-Williams, C. Fensch, and S. Moore, "A Communication Characterisation of Splash-2 and Parsec," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[4] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Annual Workshop on Modeling, Benchmarking and Simulation*, 2009.

[5] S. Bokhari, "On the Mapping Problem," *IEEE Transactions on Computers*, vol. C-30, no. 3, 1981.

[6] S. Borkar and A. A. Chien, "The Future of Microprocessors," *Communications of the ACM*, vol. 54, no. 5, 2011.

[7] B. Brandfass, T. Alrutz, and T. Gerhold, "Rank reordering for MPI communication optimization," *Computers & Fluids*, Jan. 2012.

[8] A. Chan, W. Gropp, and E. Lusk, "User's Guide for MPE Extensions for MPI Programs," 1998.

[9] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn, "MPIPP: An Automatic Profile-guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters," in *International Conference on Supercomputing*, 2006.

[10] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing Replication, Communication, and Capacity Allocation in CMPs," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2, May 2005.

[11] J. Corbet, "Toward better NUMA scheduling." [Online]. Available: http://lwn.net/Articles/486858/

[12] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[13] M. Diener, E. H. M. Cruz, and P. O. A. Navaux, "Communication-Based Mapping Using Shared Pages," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2013.

[14] M. Diener, F. L. Madruga, E. R. Rodrigues, M. A. Z. Alves, and P. O. A. Navaux, "Evaluating Thread Placement Based on Memory Access Patterns for Multi-core Processors," in *IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2010.

[15] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "Understanding Cache Hierarchy Contention in CMPs to Improve Job Scheduling," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[16] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2004.

[17] B. Goglin and S. Moreaud, "KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, Feb. 2013.

[18] T. Hoefler and M. Snir, "Generic topology mapping strategies for large-scale parallel architectures," in *International Conference on Supercomputing (ICS)*, 2011.

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

[20] ——, "Using KMP_AFFINITY to create OpenMP thread mapping to OS proc IDs," 2012. [Online]. Available: http://software.intel.com/en-us/articles/using-kmp-affinity-to-create-openmp-thread-mapping-to-os-proc-ids

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

[22] C. Karlsson, T. Davies, and Z. Chen, "Optimizing Process-to-Core Mappings for Application Level Multi-dimensional MPI Communications," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2012.

[23] D. Li, B. R. de Supinski, M. Schulz, K. Cameron, and D. S. Nikolopoulos, "Hybrid MPI/OpenMP Power-Aware Computing," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2010.

[24] H. Löf and S. Holmgren, "affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System," in *International Conference on Supercomputing*, 2005.

[25] C. Luk, R. Cohn, R. Muth, and H. Patil, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[26] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifer, B. Lucas, J. Kepner, J. Mccalpin, D. Bailey, D. Takahashi, J. Jack, and R. Rabenseifner, "Introduction to the HPC Challenge Benchmark Suite," 2005.

[27] J. Marathe and F. Mueller, "Hardware Profile-guided Automatic Page Placement for ccNUMA Systems," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[28] J. Marathe, V. Thakkar, and F. Mueller, "Feedback-Directed Page Placement for ccNUMA via Hardware-generated Memory Traces," *Journal of Parallel and Distributed Computing*, vol. 70, no. 12, 2010.

[29] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why On-Chip Cache Coherence is Here to Stay," *Communications of the ACM*, vol. 55, no. 7, Jul. 2012.

[30] G. Mercier and E. Jeannot, "Improving MPI Applications Performance on Multicore Clusters with Rank Reordering," in *European MPI Users' Group conference on Recent advances in the message passing interface (EuroMPI)*, 2011.

[31] F. Pellegrini, "Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs," in *Scalable High-Performance Computing Conference (SHPCC)*, 1994.

[32] M. J. Rashti, J. Green, P. Balaji, A. Afsahi, and W. Gropp, "Multi-core and Network Aware MPI Topology Functions," in *Recent Advances in the Message Passing Interface*, 2011.

[33] C. P. Ribeiro, J.-F. Mehaut, A. Carissimi, M. Castro, and L. G. Fernandes, "Memory Affinity for Hierarchical Shared Memory Multiprocessors," in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2009.

[34] M. M. Tikir and J. K. Hollingsworth, "Hardware monitors for dynamic page migration," *Journal of Parallel and Distributed Computing*, vol. 68, no. 9, Sep. 2008.

[35] J. Torrellas, "Architectures for extreme-scale computing," *IEEE Computer*, vol. 42, no. 11, 2009.

[36] F. Trahay, F. Rue, M. Faverge, Y. Ishikawa, R. Namyst, and J. Dongarra, "EZTrace: a generic framework for performance analysis," in *International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2011.

[37] R. F. Van der Wijngaart and H. Jin, "NAS Parallel Benchmarks, Multi-Zone Versions," 2003.

[38] 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 *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2012.