Optimizing memory affinity with a hybrid compiler/OS approach

Matthias Diener
Informatics Institute – UFRGS
mdiener@inf.ufrgs.br

Eduardo H. M. Cruz
Informatics Institute – UFRGS
ehmcruz@inf.ufrgs.br

Marco A. Z. Alves
Department of Informatics – UFPR
mazalves@inf.ufpr.br

Edson Borin
Institute of Computing – Unicamp
edson@ic.unicamp.br

Philippe O. A. Navaux
Informatics Institute – UFRGS
navaux@inf.ufrgs.br

ABSTRACT

Optimizing the memory access behavior is an important challenge to improve the performance and energy consumption of parallel applications on shared memory architectures. Modern systems contain complex memory hierarchies with multiple memory controllers and several levels of caches. In such machines, analyzing the affinity between threads and data to map them to the hardware hierarchy reduces the cost of memory accesses. In this paper, we introduce a hybrid technique to optimize the memory access behavior of parallel applications. It is based on a compiler optimization that inserts code to predict, at runtime, the memory access behavior of the application and an OS mechanism that uses this information to optimize the mapping of threads and data. In contrast to previous work, our proposal uses a proactive technique to improve the future memory access behavior using predictions instead of the past behavior. Our mechanism achieves substantial performance gains for a variety of parallel applications.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**;
• **Software and its engineering → Main memory**;

KEYWORDS

NUMA, data mapping, thread mapping, memory affinity

ACM Reference format:
Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Edson Borin, and Philippe O. A. Navaux. 2017. Optimizing memory affinity with a hybrid compiler/OS approach. In *Proceedings of CF'17, Siena, Italy, May 15-17, 2017,* 9 pages.
DOI: http://dx.doi.org/10.1145/3075564.3075566

This work received funding from the EU H2020 Programme and from MCTI/RNP-Brazil under the HPC4E project, grant agreement no. 689772, as well as from CNPq/CAPES.

CF'17, Siena, Italy

978-1-4503-4487-6/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3075564.3075566

1 INTRODUCTION

The rising number of cores in shared memory architectures has led to a higher complexity of the memory hierarchy. In modern systems, this hierarchy consists of several cache levels that are shared by different cores, as well as a Non-Uniform Memory Access (NUMA) behavior [14] due to multiple memory controllers in the system, which form a NUMA node each. In such architectures, choosing where to place threads and memory pages in the hardware affects the performance and energy consumption of memory accesses [29]. Most previous mechanisms reduce the cost of memory accesses by increasing their *locality* [11, 21]. Recent research also improves their *balance* to avoid overloading NUMA nodes [8, 10].

Locality and balance of memory accesses can be improved in two ways, which we call affinity-based *thread mapping* and *data mapping*. In thread mapping, threads that access the same data are mapped in such a way to the hardware hierarchy that they can benefit from shared caches, improving cache utilization and reducing the number of cache misses [33]. In data mapping, memory pages are mapped to the NUMA nodes such that the number of remote memory accesses is reduced [28] or the load on the memory controllers is balanced [1]. Fewer cache misses and less data movement also increase energy efficiency. Thread and data mapping affect each other and can be combined for optimal improvements [11].

Previous work uses sampling to collect information about the memory access behavior [4, 8, 11], limiting the accuracy, or uses memory tracing [10, 21], which has a high overhead and is not able to handle dynamic behavior. Other work does not handle both thread and data mapping [8], or does not take multiple applications into account [25]. Several proposals require special hardware support to determine application behavior [5, 8, 31] or modify application source code to perform the mapping [28]. Most mechanisms do not have prior information about the behavior and need to include training phases, during which opportunities for gains are lost, apart from a runtime overhead.

In this paper, we propose a hybrid approach to optimize the memory placement of parallel applications running on shared memory architectures. It combines the high accuracy and low overhead of automatic memory access prediction through a compiler-based analysis with the flexibility and generality of an OS-based mapping. Our proposal consists of two parts, which are performed during application execution. First, the memory access behavior of the application is predicted via compiler-inserted code with a high accuracy and negligible overhead. This code predicts the access behavior of parallel loops based on their actual input data during

execution. Second, the predicted behavior is passed to the OS via a system call, which performs the thread and data mapping.

To the best of our knowledge, this is the first hybrid mechanism to support both thread and data mapping. Our approach has several important advantages compared to previous work. First of all, it requires no hardware changes or specific hardware counters, is fully automatic and needs no external information about the application behavior, either via traces or source code changes/annotations. It is *proactive* instead of *reactive*, because mapping decisions are guided by predicted behavior, not by observed behavior in the past, which allows it to perform earlier and more accurate mapping decisions with a lower overhead than reactive mechanisms. We evaluate our proposal with a set of parallel benchmarks with different memory access behaviors, showing that it outperforms previous techniques and achieves results close to an Oracle-based mapping.

2 RELATED WORK

We divide previous research on affinity-based mapping into three broad groups according to where the mapping is performed: OS-based mechanisms, compiler or runtime library-based mechanisms, and mechanisms at the hardware level.

Traditional OS-based data mapping strategies for NUMA architectures are *first-touch*, *interleave* and *next-touch*. In the first-touch policy [22], a page is allocated on the first NUMA node that performs an access to the page. This is the default policy for most current operating systems. An interleave policy distributes memory pages equally between nodes to balance the load on the memory controllers. Next-touch policies mark pages such that they are migrated to the node that performs the next access to them [17].

Improved policies focus mostly on refining the data mapping during execution by using online profiling. LaRowe et al. [15] propose page migration and replication policies for early NUMA architectures. Verghese et al. [32] propose similar page migration and replication mechanisms, but use cache misses as a metric to guide mapping decisions. Recent Linux kernels include the NUMA Balancing technique [4], which uses page faults of parallel applications to detect memory accesses and perform a sampled next-touch strategy. A previous proposal with similar goals was AutoNUMA [3]. 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. Extra page faults are used to increase detection accuracy.

Current research uses a history of memory accesses to limit unnecessary migrations and perform thread mapping. An example of such a technique is kMAF [9, 11], which uses page faults to determine data sharing and page accesses, and uses this information for thread and data mapping. The Carrefour mechanism [8, 13] uses Instruction-Based Sampling (IBS), available on recent AMD architectures, to detect the memory access behavior. Pages are migrated or replicated depending on the access pattern to it. Awasthi et al. [1] propose balancing the memory controller load by migrating pages from overloaded controllers to less loaded ones. They estimate load from row buffer hit rates gathered through simulation.

Compiler and runtime-based mapping mechanisms only have knowledge about a single application, and their mapping decisions can interfere between them if there are several applications running at the same time. Majo et al. [19] 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. SPM [25] uses a similar characterization technique as our proposal, but performs the mapping in a support library. SPM has only a partial view of the application, on the single thread level. Pages are migrated to the thread that is predicted to access them first. No thread mapping or balancing operations are performed.

Nikolopoulos et al. [24] propose an OpenMP library that gathers information about thread migrations and memory statistics of parallel applications to migrate pages between NUMA nodes. Libraries that support NUMA-aware memory allocation include libnuma and MAi [28]. With these libraries, data structures can be allocated according to the specification of the programmer. 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 [27], optionally uses a source code preprocessor to statically determine data mapping policies for arrays.

Hardware-based mapping mechanisms use hardware counter statistics for mapping decisions. Marathe et al. [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. Tikir et al. [30, 31] use UltraSPARC III hardware counters to provide information for data 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) [5] 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. Intense Pages Mapping (IPM) [7] also gathers information from TLB misses, but can be implemented natively in machines with software-managed TLBs, such as Intel Itanium. However, in more common architectures with hardware-managed TLBs, IPM also requires hardware changes to access TLB contents.

We conclude that our hybrid proposal is the first mechanism to have all the following important properties: (1) it handles thread and data mapping jointly, (2) it has prior information about the memory access behavior, increasing its potential for performance gains, (3) its mapping is based on the kernel level, supporting multiple applications executing at the same time, (4) it requires no changes to applications or the hardware, (5) it supports policies that focus on locality or balance, (6) it requires no previous execution to profile the application. We will compare the gains of our proposal to several of the techniques presented in this section.

3 PREDICTING MEMORY ACCESS BEHAVIOR VIA LOOP INSTRUMENTATION

Predicting memory access behavior is the critical step for thread and data mapping, since incorrect information might result in wrong migration decisions and therefore performance losses. In most previous automatic mechanisms, the prediction is performed with an

```
1 for (i=tid*(N/T); i<(tid+1)*(N/T); i++)
2     A[i] = A[i] + B[i];
```

(a) Original code.

```
1  // start, end, and number of accesses of A:
2  void* A_start = &A[(tid)*(N/T)];
3  void* A_end = &A[(tid+1)*(N/T)-1];
4  unsigned long A_accesses = 2*(N/T);
5  // start, end, and number of accesses of B:
6  void* B_start = &B[(tid)*(N/T)];
7  void* B_end = &B[(tid+1)*(N/T)-1];
8  unsigned long B_accesses = (N/T);
9  // Inform OS of the new access patterns:
10 access_pattern(A_start, A_end, A_accesses);
11 access_pattern(B_start, B_end, B_accesses);
12 for (i=tid*(N/T); i<(tid+1)*(N/T); i++)
13     A[i] = A[i] + B[i];
14 // Inform OS to clear patterns:
15 delete_patterns();
```

(b) Code with instrumentation.

Figure 1: Example code of a parallel vector addition.

online profiling mechanism [4, 8, 11], in which the system continuously monitors the application's memory access behavior and estimates the future behavior based on past information. For example, if the mechanism detects many remote memory accesses per second on a set of pages, it may predict that this behavior will continue in the future and trigger the migration of these pages.

Online profiling suffers from three main problems that limit the performance gains that can be achieved. (1) The past behavior may not indicate how the system will behave in the future. For example, a loop that is currently executing is shaping the memory access pattern, but it is about to finish. (2) The profiling mechanism adds an extra performance overhead, for instance by adding extra page faults to monitor memory accesses. (3) The mechanism requires the execution to be profiled for some time before making a prediction, during which opportunities for improvements are lost.

To avoid these issues and perform an accurate, low overhead prediction of memory access behavior, we leverage a technique proposed by Piccoli et al. [25] to instrument the application code so that it provides hints during execution regarding the memory access behavior of loops before entering them. Such a technique has a high potential, since loop codes usually perform the majority of memory accesses and determine the memory access pattern [31].

The prediction is divided into two steps. First, the compiler analyzes the application and inserts code at the outer loops' pre-headers to compute the arrays' footprints and access frequencies as a function of program variables. Second, at runtime, the inserted code computes the footprint and the access frequency of arrays accessed inside the loop from the actual execution parameters and provides this information to the OS via system calls.

Figure 1 illustrates the code transformation performed by the compiler in a parallel vector addition. For each vector that is accessed inside the loops (for example, vector `A`), the compiler performs a symbolic range analysis on the expressions used to index the vector elements and produces code to compute the first and last positions of the vector that are accessed during loop execution. Each thread has a different identifier, stored in the `tid` variable. Since `A` is accessed using variable `i` and the symbolic range analysis of `i` indicates that it may assume values in the range from $tid \times N/T$ to $(tid + 1) \times N/T$, the first (`A_start`) and the last (`A_end`) positions of `A` that can be accessed during the loop execution are `&A[tid*(N/T)]` and `&A[(tid+1)*(N/T)-1]`, respectively.

The compiler also uses symbolic range analysis to estimate loop trip counts and produces expressions that compute the access frequency of each vector (for instance, `A_accesses`). As an example, `A` is accessed $2 \times N/T$ times per thread, since it is accessed twice (one read and one write operation) in the body of the inner loop and the loop body itself is executed N/T times per thread according to trip count estimates. After inserting code to compute the first and last positions of the vector and its access frequency, the compiler inserts code to call the `access_pattern` system call before the loop body and code to call the `delete_patterns` system call after the loop body, to provide the access patterns to the OS.

Note that at compilation time, the compiler does not know the values of `N`, `T` and `tid`, hence, estimating the loops' trip counts and the range of the vector accesses at compile time is not possible. However, by means of a symbolic range analysis, the compiler is capable of producing simple expressions that compute the trip counts and vector access ranges as a function of `N`, `T` and `tid`. These expressions are embedded into the code to be evaluated at runtime, when the values of these variables are known.

Even though we use a simple example to illustrate how the system works, the compiler analysis can capture complex memory access patterns and has shown a very high accuracy with a wide range of applications [25]. Some highly irregular or highly dynamic access patterns might not be captured accurately, however, such patterns are usually not sensitive to mapping and would not generate a performance loss from our mechanism. Since the prediction is performed at runtime, it takes changes in the program behavior into account, such as changing the input parameters of an application.

Note that there is no need to know the data partitioning among threads at compile time. The symbolic analysis produces code that will infer, for each thread, the data that may be accessed by the threads at runtime. As the system call is executed by each thread, the kernel needs to limit the number of migrations that are performed, which will be explained in the next section. We implemented our proposal in the clang compiler [16]. Since clang only recently gained support for OpenMP, we use Pthreads in our implementation, but the concepts of our proposal can be ported to other parallel APIs such as OpenMP with a moderate effort.

4 THE KERNEL MAPPING MECHANISM

We implemented the mapping part of our proposed mechanism in the Linux kernel. The mapping receives information provided by the compiler-inserted code, analyzes it, and combines it with information of the hardware topology that is collected at the kernel level to calculate optimized mappings. In this section, we describe the interface between the compiler and kernel and present how the kernel performs thread and data mapping.

Algorithm 1: `access_pattern`: Implementation of the system call in the kernel.

```
   Input: addr_start, addr_end, accesses
   GlobalData: hTable
1  begin
2      for addr ← addr_start ; addr ≤ addr_end ; addr += PAGESIZE do
3          pageInfo ← findPageInfo(hTable, addr);
4          SetAsSharer(pageInfo, tid);
5      accesses_per_page ← accesses / ((addr_end − addr_start) /
         PAGESIZE);
6      ThreadMapping(addr_start, addr_end, accesses_per_page);
7      DataMapping(addr_start, addr_end, accesses_per_page);
```

4	0	0	0	1	
3	0	0	1		1
2	0	1		1	0
1	1		1	0	0
0		1	0	0	0
	0	**1**	**2**	**3**	**4**

Figure 2: Sharing matrix of an application with 5 threads. Cells contain the amount of sharing between thread pairs.

4.1 Retrieving information from user space

The interface between the compiler and kernel consists of two system calls. The `access_pattern` call is shown in Algorithm 1. The system call receives the memory access information from user space, including the memory address range from `addr_start` to `addr_end`, and the estimated amount of memory accesses `accesses` in the range. The thread ID `tid` is determined by the kernel as it knows which thread performed the call.

We store the information provided by the call in a hash table, which is indexed by the page address. This is done in line 2, where we iterate over all pages of the memory address range provided by the call. For each page, we fetch its information from the hash table (line 3) and set the thread passed in the system call as a sharer of the page (line 4). We calculate the number of memory accesses per page in line 5. After storing the data, we perform the thread and data mapping, which are explained in the following subsections.

The `delete_patterns` system call removes information about the access patterns from the mechanism in order to indicate that a new program phase starts and the access pattern might change.

4.2 Performing the thread mapping

To analyze the sharing pattern between the threads, the memory address ranges of all threads are compared to each other. Whenever there is an overlap of memory addresses of different threads, we consider this area as shared. We store the amount of data shared between each pair of threads in a *sharing matrix*. For each shared area between threads x and y, we increment the sharing matrix in the cells (x, y) and (y, x).

To illustrate how the data sharing is detected, consider the following example. A vector of N elements in the main memory is accessed in a parallel block by T threads. Each thread i accesses the memory range from $i \times N/T$ to $(i + 1) \times N/T$. The border of each domain is shared between consecutive threads. Therefore, every time our algorithm detects the overlapping memory area of threads i and $i + 1$, we increment the sharing matrix in cells $(i, i + 1)$ and $(i + 1, i)$. Figure 2 shows the sharing matrix for this application, where cells indicate the amount of data sharing.

Algorithm 2: `ThreadMapping`: Detects the data sharing and generates a mapping of threads to cores.

```
   Input: addr_start, addr_end, accesses
   GlobalData: hTable, hardwareTopology, shMatrix[][], oldFriends[],
               nThreads
1  begin
2      for addr ← addr_start ; addr ≤ addr_end ; addr += PAGESIZE do
3          pageInfo ← findPageInfo(hTable, addr);
4          for j ← 0 ; j < pageInfo.nSharers ; j++ do
5              shMatrix[tid][pageInfo.sharers[j]] += accesses;
6              shMatrix[pageInfo.sharers[j]][tid] += accesses;
7      for i ← 0 ; i < nThreads ; i++ do
8          friends[i] ← argmax(shMatrix[i]);
9      for changed ← 0, i ← 0 ; i < nThreads ; i++ do
10         changed ← changed + (friends[i] ≠ oldFriends[i]);
11     if changed > 1 then
12         map ← EagerMap(shMatrix, hardwareTopology);
13         MigrateThreads(map);
14         oldFriends ← friends;
```

After updating the sharing matrix, we evaluate if it changed enough to perform a migration of threads, in order to reduce the overhead of the mapping algorithm and unnecessary migrations. We first verify which thread communicates most to each thread of the parallel application, which we call the *friend* thread. Afterwards, we calculate how many threads changed their friend thread compared to the previous system call. If the number of these is higher than 1, we call the thread mapping algorithm. Otherwise, no thread migration is performed.

When calling the mapping algorithm, the sharing matrix is combined with information about the memory hierarchy, which is available in the kernel, to calculate the thread mapping. We use the EagerMap algorithm [6], which has a very low overhead on the running application and scales well up to thousands of threads. EagerMap receives as input the sharing matrix and a graph representation of the memory hierarchy, in which the vertices represent cores, caches, and NUMA nodes, and the edges the links between the components. EagerMap outputs a thread mapping in which threads that share a lot of data are mapped to cores nearby in the memory hierarchy.

Algorithm 2 shows how the thread mapping procedure is implemented. The outer loop in line 2 iterates over all pages of the memory address range provided in the system call. For each page, we iterate over the threads that accessed the page (loop in line 4), and increment the sharing matrix as previously explained (lines 5–6). Afterwards, we analyze if the sharing pattern changed (lines 7–11) and then call EagerMap to calculate the thread mapping (line 12) and migrate the threads (line 13).

4.3 Performing the data mapping

To perform the data mapping, we need to differentiate the cases in which the memory address range provided in the system call has or

not been accessed before. In case the page has been accessed before, we migrate the page to another node if necessary. In case the page has not been accessed before, the page still does not physically exists in the main memory due to the demand paging technique. Therefore, we store the destination NUMA node of the pages that have not been accessed before in the hash table. When a page is accessed for the first time, we check if there is an entry in the hash table for the corresponding page. If there is an entry, we allocate the page in the NUMA node of the thread stored in the hash table. Otherwise, the destination NUMA node is selected using the native policies of the kernel.

A detailed description of the data mapping procedure can be found in Algorithm 3. As previously, the outer loop in line 2 iterates over all pages of the memory address range listed in the system call. For each page, we first evaluate if the page is suitable for a locality based mapping. In order to do that, we calculate the exclusivity level of the page [11]. The exclusivity level corresponds to the highest number of memory accesses to the page from a single thread in relation to the number of accesses from all threads (lines 4–14).

After calculating the exclusivity, we compare it against a threshold called *localityThreshold* (line 15). If the exclusivity is greater than the threshold, we use a locality based data mapping, by setting the destination node as the node who most access the page. Otherwise, we use an interleaved page mapping, in which the node of the page is determined from the lowest significant bits of the page address. This threshold is important in case a page is accessed by multiple threads, that is, if multiple system calls provide information regarding the same page. For such shared pages, the threshold limits the number of migrations.

We experimented with several values for the *localityThreshold*, between 0.7 and 0.95. The performance improvements were not very sensitive to the particular value in this range, and we experiment with a value of 0.85. Finally, we verify if the page has been accessed before (line 20). If it does, we migrate the page to the destination node. Otherwise, we store the destination node in the hash table to use it when the page is accessed for the first time, removing the need for migrating the page.

Algorithm 3: `DataMapping`: Analyzes the page usage and performs the data mapping.

```
   Input: addr_start, addr_end, accesses
   GlobalData: hTable, nNodes
1  begin
2      for addr ← addr_start ; addr ≤ addr_end ; addr += PAGESIZE do
3          pageInfo ← findPageInfo(hTable, addr);
4          for j←0 ; j<nNodes ; j++ do
5              access[j] ← 0;
6          for j←0 ; j<pageInfo.nSharers ; j++ do
7              access[ nodeOfThread(pageInfo.sharers[j]) ] += accesses;
8          max ← 0;
9          sum ← access[0];
10         for j←1 ; j<nNodes ; j++ do
11             sum ← sum + access[j];
12             if access[j] > access[max] then
13                 max ← j;
14         excl ← access[max] / sum;
15         if excl > localityThreshold then
16             node ← max;
17         else
18             node ← addr mod nNodes;
19         if CurrentNode(addr) ≠ node then
20             if PageAllocated(addr) then
21                 MigratePage(addr, node);
22             else
23                 StorePageFirstTouch(addr, node);
```

4.4 Supporting multiple applications

One of the main advantages of performing the mapping in the kernel is the fact that multiple concurrently executing applications can be taken into account when performing the thread and data mapping, as the kernel has the complete view of the system. We implemented a simple but efficient technique to support the execution of multiple applications at the same time. The main challenge lies in the fact that the virtual addresses of different applications might overlap, which would mean that one application might affect the thread or data mapping of another. To overcome this issue, we also save the ID of the process that the page belongs to in the hash table, such that we can differentiate between pages of different processes that have the same virtual address.

The same thread and data mapping algorithms are employed for the parallel applications. As there is only a single sharing matrix for the entire system, the elements of the sharing matrix corresponding to pair of threads from different processes would be zero. Hence, threads from the same processes are always mapped closer than threads from different processes, which is the desired result. The data mapping is calculated for each page separately, as before, and we differentiate between different applications via the process ID. In this way, our mapping policies can take contention on the whole system into account without the need for a special support.

4.5 Complexity of our proposal

The compiler overhead is determined by the number of data structures, as instrumentation code is inserted on a per-structure and per-loop basis. For each data structure and loop, the added code has a constant time complexity. The system call stores information about memory accesses for each page, resulting in a time complexity of $O(P)$, where P represents the number of pages the data structure uses. The complexity of the thread mapping is determined by the EagerMap algorithm, which has a time complexity of $O(T^3)$, where T represents the number of threads. The data mapping algorithm has a time complexity of $O(P)$, considering that the number of NUMA nodes and sharers is constant, resulting in a final complexity of $O(T^3 + P)$.

5 EXPERIMENTAL EVALUATION

We compare our proposed hybrid mapping mechanism to the current state-of-the-art solutions with a set of parallel benchmarks on two machines. This section begins with a description of the experimental methodology, followed by experiments with a single

running parallel application. We also evaluate our technique with multiple applications that are executing concurrently.

5.1 Methodology of the experiments

This section discusses the machines, benchmarks, and mapping mechanisms that were used in our evaluation.

5.1.1 Machines. Experiments were performed on two NUMA machines, *Xeon* and *Opteron*, which have 4 and 8 NUMA nodes, respectively. *Xeon* consists of 4 Intel Xeon X7550 processors. Each processor has its own memory controller and contains 8 cores. Each core can execute 2 threads at the same time via Simultaneous MultiThreading (SMT). The *Opteron* machine is an example of a NUMA architecture with multiple memory controllers per chip. It consists of 4 AMD Opteron 6386 processors, each with 2 memory controllers and 8 cores. It also supports execution of 2 threads per core. An overview of the architectures, including their NUMA factors calculated with the Lmbench benchmark [23], is shown in Table 1. Since both machines can execute 64 tasks at the same time, we run all parallel applications with 64 threads.

5.1.2 Benchmarks. We evaluate our proposal with a set of parallel benchmarks parallelized with Pthreads. Our baseline for the experiments consists of six applications with widely different memory access behaviors. We evaluate four applications from the linear algebra domain: matrix multiplication, matrix addition, Cholesky decomposition, and LU decomposition. We also use an implementation of the K-Nearest Neighbors (KNN) data-mining algorithm and a parallel bucket sort implementation. The memory usage of the applications varies between 12 MB (KNN) and 27 GB (matrix addition), creating different sensitivities to mapping.

We verified that our proposal works correctly with all applications from two common Pthreads-based benchmark suites, SPLASH-2 [34] and PARSEC [2]. However, most of these applications are not sensitive to mapping [12, 26]. For this reason, we selected the two benchmarks that are most sensitive to mapping, Streamcluster and Ocean_cp, from these suites. Streamcluster is an example of an application with a highly dynamic memory access behavior. We use the splash2x implementation of Ocean_cp contained in recent releases of the PARSEC suite. Both Streamcluster and Ocean_cp were executed with the *native* input size.

5.1.3 Mapping mechanisms. We compare the following mapping mechanisms: *OS*, *Compact*, *Interleave*, *Oracle*, *NUMA Balancing*, *kMAF*, *Carrefour*, *SPM*, and *Hybrid*.

Table 1: Overview of the two systems used in the evaluation.

	Property	Value
Xeon	NUMA	4 nodes, 1 NUMA node/processor, NUMA factor 1.5
	Processors	4× Intel Xeon X7550, 2.0 GHz, 8 cores, 2-way 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 NUMA nodes/processor, NUMA factor 2.8
	Processors	4× AMD Opteron 6386, 2.8 GHz, 8 cores, 2-way CMT
	Caches	8× 16 KB+64 KB L1, 8× 2 MB L2, 2× 6 MB L3
	Memory	128 GB DDR3-1600, page size 4 KB

The Linux *OS* forms the baseline of our experiments. We run an unmodified kernel, version 3.13, and use its default first-touch page allocation policy and the Completely Fair Scheduler (CFS).

The *Compact* thread mapping is a simple policy to improve memory affinity by placing threads with neighboring IDs (such as threads 0 and 1) close in the memory hierarchy, such as on cores sharing a cache. Data mapping is performed via a first-touch policy.

The *Interleave* data mapping policy, available via the `numactl` tool, distributes pages equally among the NUMA nodes according to the page address, and is a simple way to improve memory access balance. Thread mapping is handled by the Linux scheduler.

The *Oracle* thread and data mapping is used to measure the theoretical improvements that can be achieved. To calculate it, we use a Pin-based [18] memory tracer to list all memory accesses of each application on a per-thread level. We then apply the same algorithms used for our proposed mechanism to calculate the mappings and store the mappings in files. These files are read by the kernel during application startup, which then performs the thread and data mapping. Since all calculations are performed before the application starts, this mechanism has no runtime overhead.

The *NUMA Balancing* data mapping mechanism [4] of kernel 3.13 uses a sampling-based next-touch migration policy. When an application causes a page fault, the page is migrated to the NUMA node on which the thread is executing. Extra page faults are inserted into the running application to increase accuracy.

kMAF [11] performs an optimized thread and data mapping during the execution of parallel applications. The mechanism has, similarly to NUMA Balancing, no prior information about application behavior and performs the analysis and migration from information gathered from page faults during execution.

The *Carrefour* mechanism [8] uses AMD IBS techniques to characterize memory access behavior and migrate/replicate pages. It requires features only available in AMD systems and is therefore only compatible with our *Opteron* machine.

The Selective Page Migration (*SPM*) technique [25] is an example of a pure user space data mapping solution. Thread mapping is handled by the Linux scheduler.

Our proposed *Hybrid* mechanism was implemented in clang, version 3.5, and the Linux kernel, version 3.13.

All experiments were performed 20 times. We show the average values (presented as gains or reductions compared to the OS mapping) as well as the standard error.

5.2 Running single applications

We compare all mapping mechanisms against each other while executing a single parallel application at a time. The streamcluster benchmark will be also discussed in more detail in Section 5.3.

5.2.1 Xeon. Figure 3 shows the performance gains compared to the OS on the *Xeon* machine. From the results, we can separate the benchmarks into two groups. Bucket sort and KNN are applications with an unstructured memory access pattern, which means that pages will get accessed by all threads in a similar way, with lots of sharing between threads. Furthermore, their memory usage is quite low, such that most data fits into the caches of *Xeon*. In these situations, mapping can only achieve small improvements compared to the OS. The matrix multiplication has a higher memory usage, but

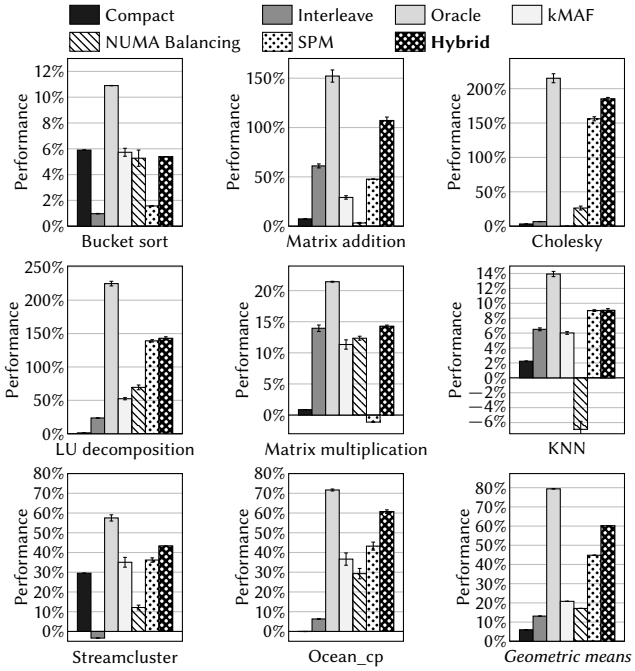

Figure 3: Performance gains on *Xeon*, normalized to the OS.

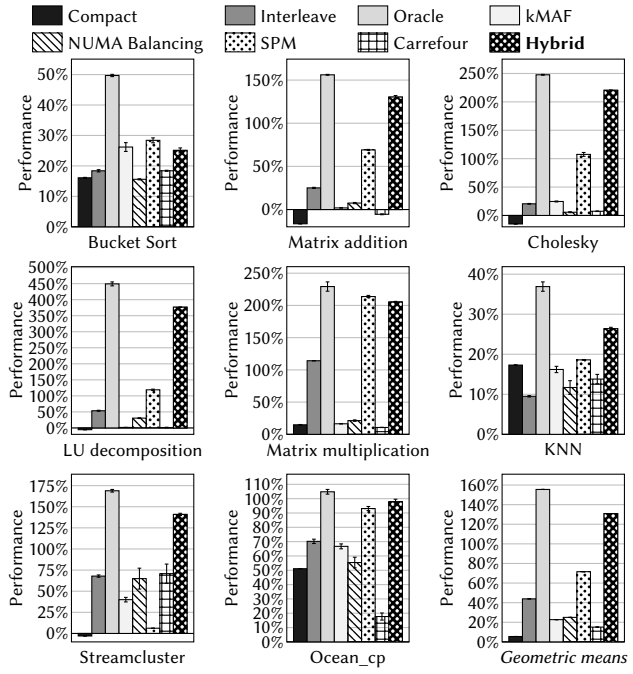

Figure 4: Performance gains on *Opteron*, normalized to the OS.

results in only small improvements from mapping, since most of the data has a good cache locality. The other benchmarks, matrix addition, Cholesky decomposition, LU decomposition, Streamcluster, and Ocean_cp, benefit more from mapping due to their structured memory access patterns and high memory usage.

The results show that thread mapping alone, as performed by the Compact mechanism, does not achieve substantial improvements compared to the OS in most cases. A data mapping-only policy like NUMA Balancing is able to reduce execution time in general, but increases it in the case of KNN slightly. Performing automatic thread and data mapping jointly, as done by kMAF, can achieve higher improvements than NUMA Balancing in some cases and avoids the performance reduction. The Oracle mechanism has no runtime overhead and therefore achieves the highest improvements. SPM and our Hybrid proposal have similar results, with slightly higher improvements for Hybrid for most benchmarks. The reason is that Hybrid also performs thread mapping and can distribute pages more equally, in contrast to SPM. It is important to mention that Hybrid is the mechanism with the closest results to the Oracle, which shows that the compiler can provide very accurate information and that prior information can result in higher improvements than when the behavior has to be detected indirectly during execution.

5.2.2 Opteron. Due to its larger number of NUMA nodes and higher NUMA factor, performance improvements on *Opteron* are generally higher than on *Xeon*. Due to its smaller cache size, even the benchmarks with a low memory usage can benefit more from mapping. As before, the simple thread clustering performed by the Compact mapping does not improve performance substantially in most cases and reduces it for some applications. Although the Interleave mapping results in higher gains than Compact, the gains are still far from the Oracle mapping, indicating that the applications are not imbalanced. kMAF and NUMA Balancing result in low improvements that are between Compact and Interleave in most cases. Carrefour is limited by the number of pages that it can characterize (30,000), and it shows performance improvements only for the benchmarks with a small memory usage. Hybrid reaches performance gains of up to 380% for LU and significantly outperforms SPM for several benchmarks.

5.2.3 Runtime overhead. To evaluate the runtime overhead, we measured the time spent in the instrumentation code and the kernel. We present the time overhead for the matrix addition, which had the highest overhead due to its high memory usage. Overhead values are 0.019 ms for the added instrumentation code, 0.746 ms for the thread mapping, and 30.1 ms for the data mapping. Values were almost identical on *Xeon* and *Opteron*. The results show that the overall overhead is determined mostly by the data mapping. Still, even in this memory-intensive application, the overhead is very low, as the data mapping of our mechanism scales linearly with memory usage. During each mapping operation only one thread is waiting, and our proposal does not stall the whole application.

5.2.4 Summary. The results of our experiments show that simple mapping policies that do not take the actual memory access behavior into account, such as Compact and Interleave, do not provide substantial performance improvements relative to the OS. Furthermore, mechanisms without prior information about the behavior, such as NUMA Balancing and kMAF, also have limited gains. By having prior information about the behavior and by applying both thread and data mapping polices jointly, as our Hybrid mechanism, the highest gains can be achieved.

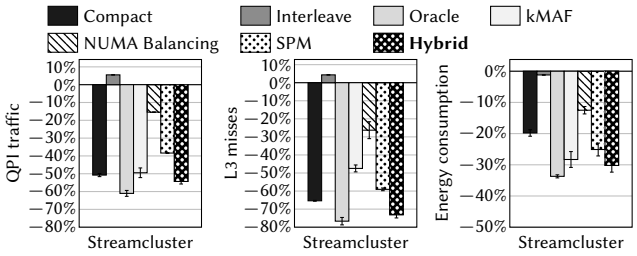

Figure 5: Streamcluster results on *Xeon*, normalized to the OS.

5.3 The Streamcluster benchmark

The Streamcluster benchmark is an example of an application with a highly dynamic memory access behavior, consisting of six phases. Parts of the input data set remain allocated and are accessed throughout the whole execution. Furthermore, Streamcluster allocates additional memory within each phase for temporary data structures. Such a behavior presents large challenges for mapping, as threads and data need to be migrated at each phase. Apart from the performance gains, we present the reduction of L3 cache misses, inter-processor QPI traffic, and system energy consumption.

Results are presented in Figure 5. Due to the relatively large caches on *Xeon*, the Compact thread mapping already results in quite high improvements, and the other mechanisms only improve slightly compared to Compact. The impact of thread mapping can be seen in the large reduction of L3 cache misses, which is higher than the reduction of QPI traffic. Energy consumption was significantly reduced as well, in a similar way as performance was improved, due to shorter execution times and a more efficient execution.

5.4 Running multiple applications

An important feature of our hybrid mechanism is that it seamlessly supports multiple parallel applications that are executing at the same time, as discussed in Section 4.4.

5.4.1 Methodology. To analyze the impact of various memory access behaviors on the results, we selected three pairs of parallel applications that showed different suitabilities for mapping on *Xeon* in our single benchmark experiments: LU+matrix addition, as examples of applications with high suitability; BucketSort+Cholesky, as a mix of applications with low and high suitability; and matrix multiplication+KNN, which showed only small performance improvements from mapping. Applications run with 64 threads each, with the same input parameters as before.

We compare three mapping mechanisms (OS, SPM, and Hybrid) with two different configurations (Sequential and Parallel). We also show results with Carrefour on *Opteron*. 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 opposite properties of the sequential case, with higher contention for resources but a more efficient usage.

5.4.2 Results. Figure 6 shows the results for the execution time (in seconds) on *Xeon*. We can see that the OS sequential case is the slowest. By running the applications in parallel, overall performance can already be improved by about 15% for the OS, as resource usage becomes more efficient. The SPM mechanism shows the opposite behavior, where the sequential case is about 10% *faster* than the parallel case for the LU+matrix addition pair, indicating that the data mappings performed by SPM of the two applications interfere with each other. As more threads are running than the machine can execute at the same time, the OS will perform more thread migrations, rendering the data mapping ineffective.

Similarly, the BucketSort+Cholesky pair also suffers a performance degradation from parallel execution with SPM. As matrix multiplication and KNN are less sensitive to mapping, SPM results in small improvements with a parallel execution. The Hybrid mechanism results in the highest improvements overall, reducing execution time by more than 50% compared to the OS-sequential case, with larger gains in the Hybrid-parallel case.

The results for the *Opteron* machine are shown in Figure 7. The general behavior is very similar to *Xeon*, with higher gains overall. SPM loses performance in the parallel case for two benchmark pairs. In parallel execution, Hybrid never hurts performance. These results show that our proposal successfully handles multiple applications. The main reason for the large improvements compared to OS and SPM lie in the fact that Hybrid performs thread mapping, which reduces the number of unnecessary migrations of threads between cores and NUMA nodes.

5.4.3 Summary. These experiments showed that mapping mechanisms that work only in user space can hurt parallel application performance due to the interference between multiple applications.

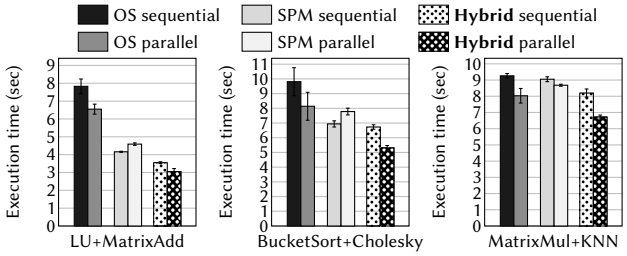

Figure 6: Running multiple applications on *Xeon*.

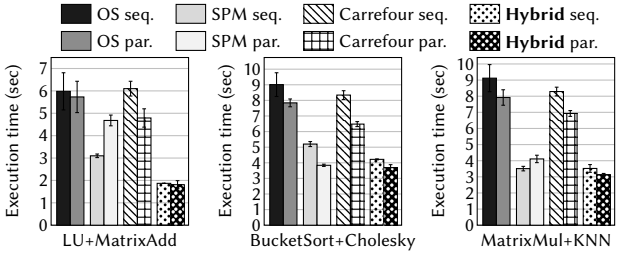

Figure 7: Running multiple applications on *Opteron*.

By performing a hybrid mapping, the performance loss can be avoided, and the performance gains are higher in several cases.

6 CONCLUSIONS

The increasing parallelism in shared memory machines has led to challenges for the memory subsystem, where the performance and energy consumption of memory accesses depend on the location of threads and the data they access. Memory accesses can be improved with thread and data mapping, which consist of an analysis of the memory access behavior of a parallel application, and a mapping policy that uses the analyzed behavior to optimize mappings.

We introduced a hybrid approach to the mapping problem, which uses a compiler extension to insert code that predicts, at runtime, the application's memory access patterns and passes it to the OS to perform thread and data mapping. Our mechanism works fully automatically during the execution of the application, and needs no changes to the application or prior information about its behavior.

We implemented our proposal in clang and the Linux kernel and performed experiments with a set of parallel applications on two NUMA machines. Results show that our proposal substantially outperforms previous work in most cases, reaching gains of up to 380%. For the future, we will extend our proposal to a wider range of parallel applications, including those that use OpenMP.

REFERENCES

[1] Manu Awasthi, David W. Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. 2010. Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[2] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[3] Jonathan Corbet. 2012. AutoNUMA: the other approach to NUMA scheduling. (2012). http://lwn.net/Articles/488709/

[4] Jonathan Corbet. 2012. Toward better NUMA scheduling. (2012). http://lwn.net/Articles/486858/

[5] Eduardo H. M. Cruz, Matthias Diener, Marco A. Z. Alves, Laércio L. Pilla, and Philippe O. A. Navaux. 2016. LAPT: A Locality-Aware Page Table for thread and data mapping. *Parallel Comput.* 54, May (2016).

[6] Eduardo H. M. Cruz, Matthias Diener, Laércio L. Pilla, and Philippe O. A. Navaux. 2015. An Efficient Algorithm for Communication-Based Task Mapping. In *International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*.

[7] Eduardo H. M. Cruz, Matthias Diener, Laercio L. Pilla, and Philippe O. A. Navaux. 2016. Hardware-Assisted Thread and Data Mapping in Hierarchical Multicore Architectures. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 3 (2016), 1–25. DOI: http://dx.doi.org/10.1145/2975587

[8] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[9] Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Philippe O. A. Navaux, Anselm Busse, and Hans-Ulrich Heiss. 2016. Kernel-Based Thread and Data Mapping for Improved Memory Affinity. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 27, 9 (2016).

[10] Matthias Diener, Eduardo H. M. Cruz, and Philippe O. A. Navaux. 2015. Locality vs. Balance: Exploring Data Mapping Policies on NUMA Systems. In *International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*.

[11] Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux, Anselm Busse, and Hans-Ulrich Heiß. 2014. kMAF: Automatic Kernel-Level Management of Thread and Data Affinity. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[12] Matthias Diener, Eduardo H. M. Cruz, Laércio L. Pilla, Fabrice Dupros, and Philippe O. A. Navaux. 2015. Characterizing Communication and Page Usage of Parallel Applications for Thread and Data Mapping. *Performance Evaluation* 88-89, June (2015).

[13] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. 2015. Challenges of memory management on modern NUMA systems. *Commun. ACM* 58, 12 (2015).

[14] Christoph Lameter. 2013. An overview of non-uniform memory access. *Commun. ACM* 56, 9 (2013).

[15] Richard P. LaRowe, Mark A. Holliday, and Carla Schlatter Ellis. 1992. An Analysis of Dynamic Page Placement on a NUMA Multiprocessor. *ACM SIGMETRICS Performance Evaluation Review* 20, 1 (1992).

[16] Chris Lattner. 2011. LLVM and Clang: Advancing Compiler Technology. In *Free and Open Source Developers European Meeting (FOSDEM)*.

[17] Henrik Löf and Sverker Holmgren. 2005. affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In *International Conference on Supercomputing (ICS)*.

[18] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[19] Zoltan Majo and Thomas R. Gross. 2012. Matching memory access patterns and data placement for NUMA systems. In *International Symposium on Code Generation and Optimization (CGO)*.

[20] Jaydeep Marathe and Frank Mueller. 2006. Hardware Profile-guided Automatic Page Placement for ccNUMA Systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[21] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. 2010. Feedback-Directed Page Placement for ccNUMA via Hardware-generated Memory Traces. *Journal of Parallel and Distributed Computing (JPDC)* 70, 12 (2010).

[22] Michael Marchetti, Leonidas Kontothanassis, Ricardo Bianchini, and Michael L. Scott. 1995. Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems. In *International Parallel Processing Symposium (IPPS)*.

[23] Larry McVoy and Carl Staelin. 1996. Lmbench: Portable Tools for Performance Analysis.. In *USENIX Annual Technical Conference (ATC)*.

[24] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. 2000. UPMLIB: A Runtime System for Tuning the Memory Performance of OpenMP Programs on Scalable Shared-Memory Multiprocessors. In *Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR)*.

[25] Guilherme Piccoli, Henrique N. Santos, Raphael E. Rodrigues, Christiane Pousa, Edson Borin, and Fernando M. Quintão Pereira. 2014. Compiler support for selective page migration in NUMA architectures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[26] Aske Plaat, Henri E. Bal, Rutger F. H. Hofman, and Thilo Kielmann. 2001. Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects. *Future Generation Computer Systems* 17, 6 (2001), 769–782.

[27] Christiane Pousa Ribeiro, Marcio Castro, Jean-François Méhaut, and Alexandre Carissimi. 2010. Improving memory affinity of geophysics applications on NUMA platforms using Minas. In *International Conference on High Performance Computing for Computational Science (VECPAR)*.

[28] Christiane Pousa Ribeiro, Jean-François Méhaut, Alexandre Carissimi, Marcio Castro, and Luiz Gustavo Fernandes. 2009. Memory Affinity for Hierarchical Shared Memory Multiprocessors. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*.

[29] John Shalf, Sudip Dosanjh, and John Morrison. 2010. Exascale Computing Technology Challenges. In *High Performance Computing for Computational Science (VECPAR)*.

[30] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2004. Using Hardware Counters to Automatically Improve Memory Performance. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[31] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2008. Hardware monitors for dynamic page migration. *Journal of Parallel and Distributed Computing (JPDC)* 68, 9 (sep 2008).

[32] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating system support for improving data locality on CC-NUMA compute servers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[33] Wei Wang, Tanima Dey, Jason Mars, Lingjia Tang, Jack W Davidson, and Mary Lou Soffa. 2012. Performance Analysis of Thread Mappings with a Holistic View of the Hardware Resources. In *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*.

[34] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: characterization and methodological considerations. In *International Symposium on Computer Architecture (ISCA)*.