LAPT: A Locality-Aware Page Table for Thread and Data Mapping

Eduardo H. M. Cruz[1], Matthias Diener[1], Marco A. Z. Alves[1], Laércio L. Pilla[2], Philippe O. A. Navaux[1]

[1] *Informatics Institute – Federal University of Rio Grande do Sul – Porto Alegre, Brazil*
{ehmcruz, mdiener, mazalves, navaux}@inf.ufrgs.br

[2] *Department of Informatics and Statistics – Federal University of Santa Catarina – Florianópolis, Brazil*
laercio.pilla@ufsc.br

Abstract

The performance and energy efficiency of current systems is influenced by accesses to the memory hierarchy. One important aspect of memory hierarchies is the introduction of different memory access times, depending on the core that requested the transaction, and which cache or main memory bank responded to it. In this context, the locality of the memory accesses plays a key role for the performance and energy efficiency of parallel applications. Accesses to remote caches and NUMA nodes are more expensive than accesses to local ones. With information about the memory access pattern, pages can be migrated to the NUMA nodes that access them (data mapping), and threads that communicate can be migrated to the same node (thread mapping).

In this paper, we present LAPT, a hardware-based mechanism to store the memory access pattern of parallel applications in the page table. The operating system uses the detected memory access pattern to perform an optimized thread and data mapping during the execution of the parallel application. Experiments with a wide range of parallel applications (from the NAS and PARSEC Benchmark Suites) on a NUMA machine showed significant performance and energy efficiency improvements of up to 19.2% and 15.7%, respectively, (6.7% and 5.3% on average).

Keywords:
Communication, Thread mapping, Data mapping, NUMA, Page table

1. Introduction

Advances in shared-memory architectures have led to a large increase in thread-level parallelism (TLP) in computer systems, caused by the increase in number of processors per system, and cores per chip. As parallel applications need to access shared data, the placement of threads and data can have a great impact on performance and energy consumption. This impact varies among different architectures, since each processor family uses a different organization for the cache hierarchy and the main memory system. The memory hierarchy is formed by several cache levels, where the levels closer to the processor cores tend to be private, followed by caches shared by multiple cores. In systems with Non-Uniform Memory Access (NUMA), the main memory is also clustered between cores or processors.

A locality-aware thread mapping can reduce the number of cache invalidations and line replications, as threads that share data will be kept close and will share the same memory resources [1]. The accesses to shared data represent communication between the threads. A locality-aware data mapping in NUMA systems can reduce the number of accesses to remote nodes, as most of the data accessed by each threads will be kept close to them [2]. Both mappings can impact different resources, such as interconnection systems and cache coherence protocols.

Mechanisms to map threads and data have been proposed. Most of the proposals perform thread or data mapping separately [1, 3, 4]. Several approaches focus on static mapping using memory traces from previous executions in controlled environments such as simulators [5], which has a high overhead and is not able to handle dynamic behavior. Some mechanisms depend on particular APIs [6], or rely on indirect or incomplete information about the locality of memory accesses [1, 7].

In this paper, we present a *Locality-Aware Page Table (LAPT)* to enable thread and data mapping by operating systems. Our mechanism detects the locality of memory accesses in hardware, and performs the mappings in software. LAPT operates during run time, allowing mappings to be performed dynamically. It does not require previous knowledge of the application's behavior, or modifications to its source code or parallelization libraries. By detecting locality in hardware, LAPT detects more memory access samples than previous mechanisms, generating more accurate information.

This paper represents an extension of our previous work [8]. We extend the paper by changing LAPT to store all the data used to calculate the data mapping in the page table, improving the efficiency of the mechanism by avoiding the access to different data structures. Furthermore, we perform an extensive analysis of how LAPT behaves under different scenarios, varying cache memory and memory page sizes, as well as interconnection latencies. We also include an evaluation of cache misses and interconnection usage inside a full system simulator, which help us understand the performance improvements.

2. Related Work

In static mapping mechanisms, the application execution is usually monitored to generate memory access traces, which are then analyzed to compute a mapping. In order to monitor memory accesses, simulators [5] or hardware performance monitoring units (PMU) can be used [2]. The main drawback of these methods is the requirement of memory traces, which are expensive to generate and analyze, and are not able to handle changes in application behavior. LAPT does not suffer these disadvantages, as it performs the mapping dynamically. PMUs can also be used to perform dynamic mapping [3], but with a high overhead. Other mechanisms that perform dynamic mapping depend on indirect performance statistics from hardware counters [9, 10], generating less accurate mappings, while LAPT knows the memory addresses accessed by each thread.

In [1], the authors use hardware counters that provide the memory addresses of requests solved by remote cache memories. This detects incomplete communication patterns because memory requests resolved by local caches are not considered. Also, data mapping was not performed, and therefore their proposal is not able to reduce remote memory accesses in NUMA architectures. kMAF [7] uses page faults of parallel applications to detect memory accesses. Carrefour [11] is a similar mechanism that uses sampling to detect page usage. Due to its overhead, the authors restrict the mechanism to 30,000 pages, which limits its use to applications with a low memory usage. Although these proposals have access to memory addresses, only a small sample of all accesses is taken into account. LAPT guarantees that all memory pages are considered when detecting locality by tracking the TLB misses.

3. LAPT – A Locality-Aware Page Table

In order to enable operating systems to perform thread and data mapping, our mechanism uses the virtual memory implementation of current architectures. Virtual memory requires the translation of virtual addresses to physical addresses for every memory access. To do so, the operating system keeps page tables in the main memory that contain the physical address and protection related information for every page. To reduce the amount of accesses to the main memory for address translation, a special cache memory called Translation Lookaside Buffer (TLB) is responsible for caching the page table translation entries for the most recently accessed pages. On every memory access, the processor checks if the corresponding page has a valid entry in the TLB. If a valid entry already exists (TLB hit), the virtual address is translated and the processor performs the memory access. In case of a TLB miss, the processor performs a page table walk and caches the entry in the TLB before proceeding with the address translation.

Our *Locality-Aware Page Table* (LAPT) implements changes to the virtual memory subsystem on both the hardware and software levels, as illustrated in Figure 1. On the hardware level, LAPT keeps track of all TLB misses, detecting the communication between threads and registering a list of the latest threads that accessed each page in fields introduced in the corresponding page table entry. On the software level, the operating system maps threads to cores based on the detected communication pattern, and maps pages to NUMA nodes by checking which threads accessed each page.

We detail LAPT in this section. We first explain how communication is detected in hardware. Afterwards, we describe how communication information is employed to map threads and data. Lastly, we discuss the overhead of LAPT.

3.1. Communication Detection

LAPT requires the addition of an entry in the page table called *communication vector* (CV) to identify the threads that access a page. Each CV stores the IDs of the last threads that accessed the corresponding page. Whenever CV gets full, an old thread ID needs to be removed to make room for the new one, keeping temporal locality. LAPT also introduces a *communication matrix* (CM) in main memory for each parallel application. It stores an estimation of the amount of communication between each pair of threads. Special registers containing the memory address and dimensions of CM, as well as the ID of the thread being executed, must be added to the architecture and updated by the operating system.

The behavior of LAPT is as follows. When thread T tries to access a page P but its entry is not present in the TLB, the processor performs a page table walk. Besides fetching the page table entry of P, the processor also fetches the corresponding communication vector CV_P. LAPT then increments the communication matrix CM in row T, for all the columns that correspond to a thread in CV_P:

$$CM[T][\,CV_P[i]\,] \leftarrow CM[T][\,CV_P[i]\,] + 1, \text{where } 0 \le i < |CV_P| \tag{1}$$

After updating CM, LAPT inserts thread T into CV_P:

$$CV_P[i] \leftarrow CV_P[i-1], \text{where } 0 < i < |CV_P| \tag{2}$$

$$CV_P[0] \leftarrow T \tag{3}$$

3.2. Thread Mapping Computation

The information provided by the communication detection is used to calculate an optimized mapping of threads to processing units (PUs) during the execution of the parallel application. As the mapping problem is NP-hard [12], the use of efficient heuristics are required to calculate the mapping. Dynamic mapping requires algorithms with a short execution time, since its overhead directly affects the executing application.

We model thread mapping as a graph problem with an application graph and a machine hierarchy graph. The application graph can be obtained directly from the communication matrix (CM) kept by LAPT. Vertices represent threads and edges represent the amount of communication between them. In the machine hierarchy graph, vertices represent the components of the memory hierarchy, such as the cores and caches, and edges represent their links.

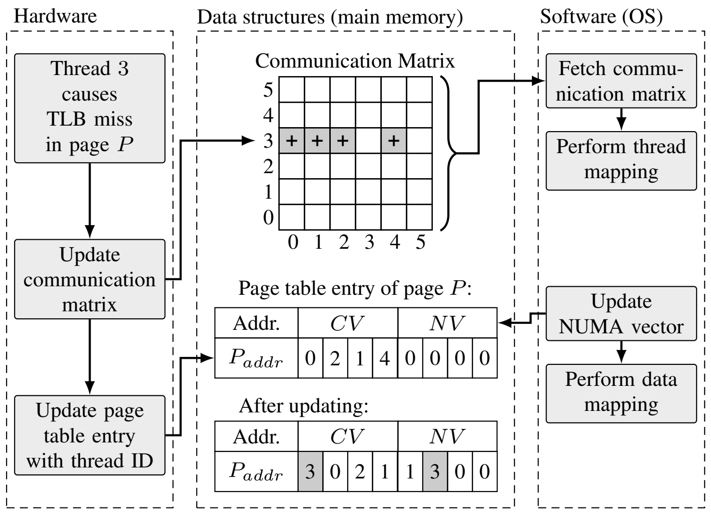

Hardware
Data structures (main memory)
Software (OS)
Thread 3 causes TLB miss in page P
Update communication matrix
Update page table entry with thread ID
Communication Matrix
5
4
3
2
1
0
0 1 2 3 4 5
+ + + +
Page table entry of page P:
Addr. CV NV
P_{addr} 0 2 1 4 0 0 0 0
After updating:
Addr. CV NV
P_{addr} 3 0 2 1 1 3 0 0
Fetch communication matrix
Perform thread mapping
Update NUMA vector
Perform data mapping

Figure 1: Overview of the LAPT mechanism.

To compute the mapping, we use the dual recursive bipartitioning algorithm of the Scotch graph library [13] version 6.0. It receives as input the application and machine hierarchy graphs, and outputs the thread mapping. Scotch calculates the minimum edge cut in the graph, such that threads that have intensive communication are kept in the same subset. It repeats this procedure recursively in each graph subset. We used Scotch because it has a short execution time (less than 1 ms to map 32 threads) while providing good results [14], therefore, being suitable for online mapping. It has a complexity of $\mathcal{O}(N^3)$, where N is the number of processes to be mapped [15]. Other libraries and algorithms, such as METIS [16], Zoltan [17], EagerMap [18] and TreeMatch [19], could be used. The algorithm receives the communication and hierarchy graphs as input and outputs the PU for each thread such that the total cost of communication is minimized. This information is then used to migrate threads to their assigned PUs.

The operating system chooses the frequency in which the thread mapping routine is called. To prevent unnecessary migrations and to reduce the overhead, we adjust this frequency dynamically. If the computed mapping does not differ from the previous mapping, we increase the mapping interval by 50 ms because the mapping is stable. If mappings differ, we halve the interval. The interval is kept between 50 ms and 500 ms to limit the overhead while still being able to react quickly to changes of communication behavior. The initial time interval was set to 300 ms in the experiments.

3.3. Data Mapping Computation

To calculate the data mapping, the operating system verifies the contents of the communication vector CV of each page in the page table. We also introduce a field in the page table to store an estimation of the amount of memory accesses from each NUMA node, which we call *NUMA vector* (NV). This field is updated by the operating system and it has one counter per NUMA node.

When the data mapping routine is called, for every page P of the application, the counters of the NUMA nodes of each thread in CV_P are incremented in the NUMA vector of this page (NV_P). This is illustrated in Equation 4, where $node(x)$ is a function that returns the NUMA node in which thread x is running.

$$NV_P[\,node(CV_P[i])\,] \leftarrow NV_P[\,node(CV_P[i])\,] + 1, \text{where } 0 \leq i < |CV_P| \quad (4)$$

After the contents of NV_P are updated, we use Equations 5 and 6 to determine if P, which currently resides in node M, should be migrated to the NUMA node N.

$$DataMap(P) = \{N \mid NV_P[N] = max(NV_P)\} \quad (5)$$

$$Migrate = \begin{cases} true & \text{if } max(NV_P) \geq 2 \cdot avg(NV_P) \text{ and } N \neq M \\ false & \text{otherwise} \end{cases} \quad (6)$$

A page migration will happen only if the value of the counter of the node returned by $DataMap$ is greater or equal to twice its average value. In this way, we reduce the possibility of having a ping-pong effect of page migrations between NUMA nodes.

Naturally, a migration happens only if the node returned by the $DataMap$ function is different from the NUMA node in which page P currently resides. Also, every time a page is migrated and the average value of NV_P is at least 1, we use an aging technique in which all elements of NV are halved, making it possible to adapt to changes in access behavior.

The operating system calls the data mapping routine in two situations: (I) whenever the thread mapping routine is called and causes a change in the mapping, in order to move the pages used by the threads when they migrate; (II) and whenever there has been a long time since the last call, since the data mapping may change even if the thread mapping remains the same. In our experiments, the maximum interval to call the data mapping routine was set to 500 ms.

3.4. Example of the Operation of LAPT

To illustrate how LAPT works, consider the example shown in Figure 1, where an application with 6 threads is executing on a NUMA machine with 4 nodes, and communication vectors support up to 4 thread IDs. Thread 3 tries to access page P but it does not have an entry in the TLB. The processor then performs a page table walk to read the corresponding page table entry. Besides reading the physical address and page protection information, it reads the communication vector, which contains thread IDs 0, 2, 1, and 4, in the order from MRU to LRU position. The core running thread 3 continues its execution. In parallel to that, LAPT increments the communication matrix in cells $(3, 0)$, $(3, 2)$, $(3, 1)$ and $(3, 4)$. After that, LAPT updates the communication vector, moving thread 3 to the MRU position and shifting all the other threads in CV towards the LRU position, removing thread 4.

During run time, the operating system evaluates the communication matrix to update the thread mapping. It changes the mapping of the threads to execute thread 0 on NUMA node 0, and migrates the other threads to node 1. Then, it evaluates the NUMA Vector of P for data mapping. The initial value of all the elements in NV_P is 0. Since only thread 0 is executing on node 0, the corresponding entry in NV_P will be incremented by 1. Likewise, the other three threads that are in CV_P are running on node 1, whose value will be incremented by 3. The node with the highest value in NV_P is node 1 (value 3) and the average of the values of NV_P is 1. Therefore, page P is migrated to node 1 following Equation 6.

3.5. Overhead

LAPT's overhead consists of storage space in main memory, circuit area to implement LAPT in the processor, and a run time overhead.

3.5.1. Memory Storage Overhead

The communication matrix (CM) requires $4 \cdot N^2$ bytes, where N is the number of threads, considering an element size of 4 Bytes. For 256 threads, the communication matrix requires 256 KByte. If the parallel application creates more threads than the maximum supported by the communication matrix during run time, the operating system can allocate a larger matrix and copy the values from the old matrix.

In the last level page table entries, we store the communication vector (CV) and NUMA Vector (NV). A common size for each page table entry in current architectures is 8 Bytes, as in x86-64 [20]. We extend the size of each entry to 16 Bytes. We reserve 8 bits to store each thread ID of CV, supporting up to 256 threads per parallel application, and track 4 threads per page. We also reserve 8 bits to store each count of NV, and support 4 NUMA nodes. The space used in the page table represent 0.2% of memory space overhead compared to the original system when using a standard 4 KByte page size. To support large systems, we would just need to use more memory.

3.5.2. Circuit Area Overhead

LAPT requires the addition of some registers, adders, and multiplexers to each core. More specifically, 64-bit registers are used to store the position in memory of CM, intermediary values to compute the memory position of an entry on CM, and the displacement to read CV for a page. 16-bit registers store the ID of the current running thread, the IDs stored on CV and one of these IDs to compute the address of an entry on CM. One 32-bit register is required to store the value of $CM[T][\ CV_P[i]\ ]$. Two 64-bit carry look-ahead adders are employed to compute the addresses of CV_P and $CM[T][\ CV_P[i]\ ]$, while a 32-bit carry look-ahead adder is used to increase the value of $CM[T][\ CV_P[i]\ ]$. Finally, multiplexers define which values are summed. In total, LAPT requires less than $25,000$ additional transistors per core, which represents an increase in transistors of less than 0.009% in a current processor.

3.5.3. Runtime Overhead

The additional hardware introduced by LAPT is not in the critical path, since it operates in parallel to the normal operation of the processor. On the hardware level, the time overhead introduced by LAPT consists of the additional memory accesses to update the page table entries and communication matrix. The amount of additional memory accesses depends on the TLB miss rate, which differs for each application. The accesses to the communication matrix do not need to be locked since each row is updated only by one thread. To update the statistics in the page table, a race condition can happen in case threads generate a TLB miss for the same page at the same time. Since the time LAPT takes to update the page table is small, this race condition is a rare event. Also, this race condition would not cause the application to fail, just a slight reduction in accuracy. On the software level, the time overhead includes the calculation of the thread and data mappings, and the respective migrations. In general, computing the mapping more frequently increases the overhead, but also increases the accuracy. The scalability of LAPT is affected very little by the number of NUMA nodes or cores. If for any reason the number of nodes or cores increases to a point that affects LAPT's overhead, the operating system could compute the mapping less frequently.

4. Experimental Evaluation

We perform experiments on a real machine and in a full system simulator. The real machine has two NUMA nodes and one Intel Xeon E5-2650 processor per node, with a total of 32 virtual cores, running version 3.8 of the Linux kernel. In this machine, the latency of remote memory accesses are 40% higher than the latency of local memory accesses. Since LAPT is an extension to current hardware, we use Pin [21], a binary instrumentation tool, to generate information regarding the thread and data mappings. We developed a kernel module that receives this mapping information and, during the execution of the application, maps the threads and data accordingly. The full system simulator used was Simics [22] extended with the GEMS-Ruby memory model [23] and the Garnet interconnection model [24], and simulated 8 cores organized in 4 NUMA nodes. As workloads, we used the OpenMP implementation of the NAS Parallel Benchmarks [25] v3.3.1 and the PARSEC benchmark suite [26] v3.0.

Detailed information about the methodology can be found in [8]. The following sections present an evaluation of the performance and energy consumption improvements of our proposal, as well as its overhead. We also explore the design space, analyzing how LAPT behaves on different environments.

4.1. Performance Results

The communication patterns of a subset of our workloads are depicted in Figure 2. Since the absolute values of the contents of the communication matrices vary significantly between benchmarks, we normalized each communication matrix to its own maximum value and color the cells according to the amount of communication. Darker cells indicate more communication.

The results obtained in Simics can be found in Figure 3. We evaluate execution time, L2 cache misses per thousand instructions (MPKI) and interchip interconnection traffic. The results obtained in Simics are from a single execution due to simulation time constraints. We compare the results on the simulator to its default mapping and to an oracle mapping. The default mapping is the original scheduler of the Linux kernel, combined with an interleaved data mapping policy of GEMS. The oracle mapping generates mappings considering all memory accesses and the memory hierarchy. We also used the Scotch library [13] in the oracle mapping, otherwise it would be unfeasible to calculate the thread mappings. Results are normalized to the default mapping.

The results obtained in the real machine are shown in Figure 4. In the real machine, we evaluate execution time, L2 and L3 cache misses per thousand instructions (MPKI) and interchip interconnection traffic. Cache misses and interchip traffic were measured using Intel PCM. All experiments in the real machine are the averages obtained from 30 executions, and we show a 95% confidence interval calculated with Student's t-distribution. In the real machine, we compare the results of our proposal to the default mapping performed by the operating system, to random static mappings and to an oracle mapping. For the random mapping, we randomly generated a thread

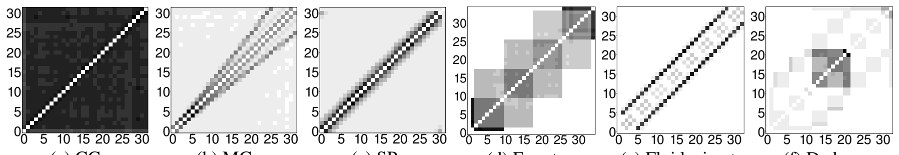
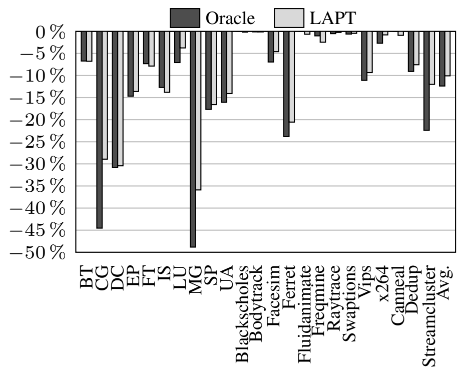


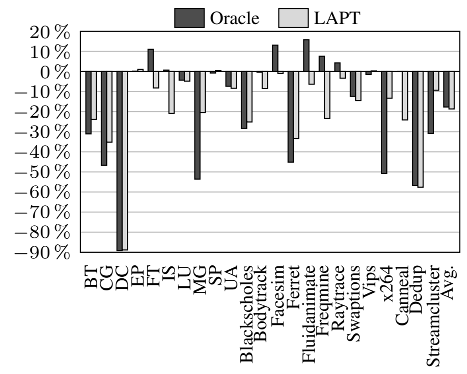
Figure 2: Communication pattern of applications with different characteristics. Axes represent thread IDs. Cells show the amount of accesses to shared pages between threads. Darker cells indicate more accesses.

and data mapping for each execution, and no migrations are performed. For the oracle mapping, we generated traces of all memory accesses for each application and perform an analysis of the communication and page usage patterns, similar to [5], and use the Scotch library [13] to calculate the thread mappings considering the memory hierarchy. The oracle mapping information is then fed to the kernel module, as explained at the beginning of Section 4. Results are normalized to the default mapping of the operating system.

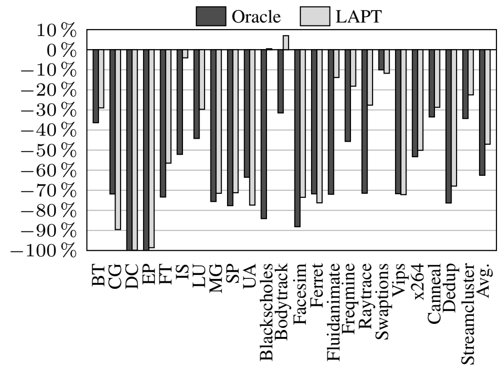
Some applications are influenced by both thread and data mapping. One example is SP. The communication pattern of SP, shown in Figure 2c, highlights that there are threads that communicate more with a small subgroup. When an application presents this characteristic, mapping threads that communicate to nearby cores in the memory hierarchy improves performance. In the case of SP, most communication happens between neighboring threads, which is very common on domain decomposition parallel applications. BT and LU have similar patterns. In applications influenced by both thread and data mapping, a locality-aware mapping usually reduces cache misses and interchip interconnection traffic.



(a) Execution time.

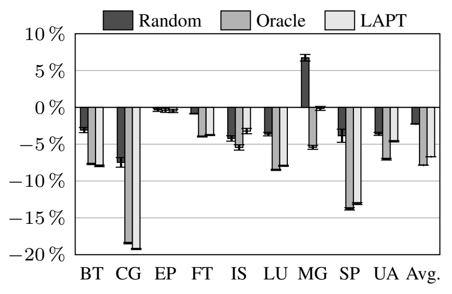


(b) L2 cache MPKI.

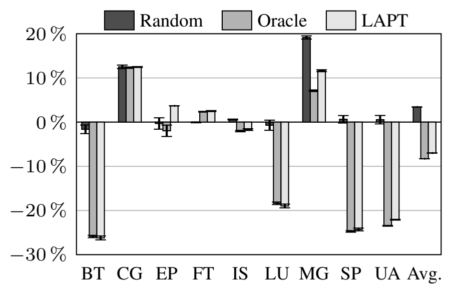


(c) Interchip interconnection traffic.

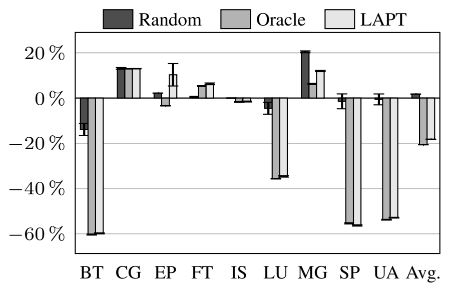
Figure 3: Performance results in Simics, normalized to the default mapping.



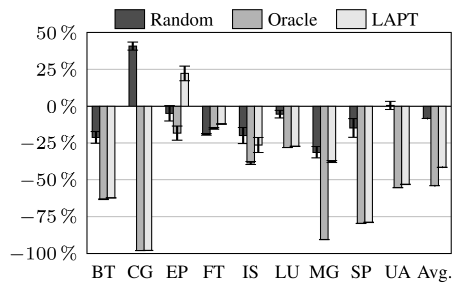
(a) Execution time.



(b) L2 cache MPKI.



(c) L3 cache MPKI.



(d) QPI interconnection traffic.

Figure 4: Performance results on the real machine, normalized to the OS.

Other communication patterns are suitable for mapping too. For instance, the communication between more distant threads in MG is more evident than in the other applications. However, in MG, only interconnection traffic was reduced in the real machine, and in Simics the reduction of interchip traffic was higher than cache misses. The cause of this is that the amount of memory used in MG is much higher than the available cache memory space, such that only a small fraction of shared data can be stored in cache. However, we can observe the importance of thread mapping in MG by the reduction of interchip traffic, since threads that communicate are mapped to cores in the same NUMA node. In Fluidanimate, the communication between more distant threads is also more evident, but the improvements obtained by LAPT over the operating system was lower.

Ferret and Dedup follow a pipeline communication model, where threads that communicate through the pipeline form communication clusters (Figures 2d and 2f). Another interesting aspect of these two applications is that they create more threads than the number of cores. Ferret created 35 threads and Dedup 27 threads, while the simulated machine has 8 cores. This proves that LAPT is able to handle applications that have more than one thread per core by adding a register in the architecture containing the identifier of the thread running in the core. In both Ferret and Dedup, cache misses and interchip interconnection were reduced significantly because LAPT was able to detect which threads communicate in the pipeline parallelization model of these applications, mapping them and their data nearby in the memory hierarchy.

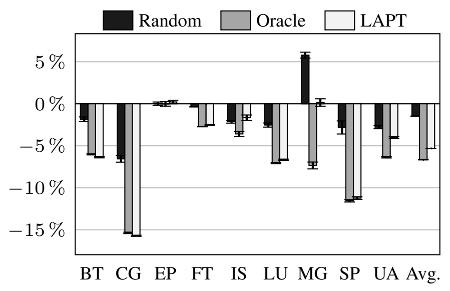
For applications whose communication pattern is such that threads present similar amounts of communication, thread mapping is not able to improve performance. CG and Vips are examples of this kind of application. This happens because the communication can not be optimized, regardless of the thread mapping. We can observe this in the pattern of CG, shown in Figure 2a, where each pair of threads has a similar amount of communication. On the other hand, the performance of these applications may still be improved by data mapping. Even if an application does not communicate much among its threads, each thread will still need to access its own private data, which can only be improved by data mapping. CG presented the highest improvement in the real machine, reducing execution time by 19.2%. It is important to note that this does not mean that data mapping is more important than thread mapping, because the effectiveness of data mapping depends on thread mapping for shared pages.

We can observe in CG that the amount of cache misses was not reduced in the real machine. In the simulator, the amount of cache misses in CG was reduced due to the lower amount of unnecessary thread migrations introduced by the operating system. As previously said, the influence of thread mapping in CG is very low, such that it is difficult to improve the shared data usage in the caches. However, the interchip interconnection traffic was reduced to almost zero in the real machine and 89.5% in Simics. This happened because data mapping migrated the private data of each thread to its NUMA node, reducing interchip traffic. Vips results behave similarly.

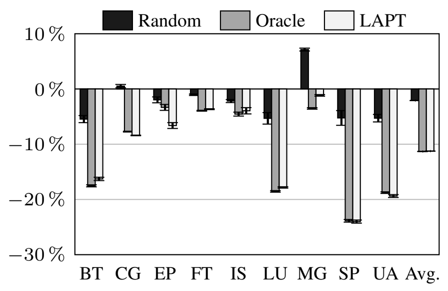
The communication pattern of Swaptions is similar to the one of Vips, not being influenced by thread mapping. Also, the memory usage of Swaptions is low, such that almost all its data fits into the caches and are thereby not affected by data mapping. EP is a CPU-bound application [25] with almost no communication between threads. Similarly to Swaptions, EP's data fits into the caches of the real machine, which makes data mapping irrelevant. However, this is not the case of EP in Simics due to the lower cache memory size, such that data mapping improved performance.

LAPT reduced the L2 and L3 MPKI in the real machine by 7.0% and 18.2% on average. The interconnection traffic was reduced by an average of 41.5% and 47.1% in the real machine and Simics, respectively. L3 MPKI had a higher reduction than L2 MPKI due to the higher cache size, which allows more caching of shared data. Execution time was reduced by 6.7% and 10.1% on average in the real machine and Simics, which is a lower reduction than the observed in cache misses and interchip interconnection traffic. The reduction of execution time tends to be a fraction of the reduction of cache misses and the interchip traffic. An improved mapping directly impacts in cache misses and interconnection traffic, which then impact execution time.

The results obtained with LAPT are similar to the oracle mapping, demonstrating its effectiveness. Occasionally, LAPT performed better than the oracle mapping. This may happen because we only consider the memory accesses to generate the oracle mapping, while there are several other factors that influence performance, such as influence from the operating system and other processes, contention in the interconnections or functional units, among others. For instance, if during the execution using the oracle mapping a thread from another process is scheduled, it will interfere with its performance. In most cases, it performed significantly better than the random mapping. This shows that the gains compared to the operating system are not due to the unnecessary



(a) Package energy.



(b) DRAM energy.

Figure 5: Energy consumption results in the real machine, normalized to the OS.

migrations introduced by the operating system, but due to a more efficient usage of the machine resources.

As explained in Section 3.5.3, LAPT introduces a performance overhead at hardware and software levels. The average performance overhead was only 0.36% and 0.4%, respectively. The time required to update the statistics during each TLB miss was on average 162 cycles (measured inside Simics/GEMS). However, application execution is not stalled by LAPT. Regarding related work, we compared LAPT to three previous techniques: Autopin [9], the Azimi thread mapping [1], and to the Marathe [3] data mapping mechanism. The results show that indirect or incomplete sources of communication information are not accurate to optimize memory access locality. Also, mechanisms that perform both thread and data mappings are able to achieve better improvements than mechanisms that perform these mappings separately. More details about the overhead and the comparison to related work are given in [8].

4.2. Energy Consumption Results

The energy consumption was measured in the real machine using the Running Average Power Limit (RAPL) hardware counters [20] with Intel PCM. Results of the total amount of processor and DRAM energy consumption are shown in Figures 5a and 5b. We can observe that the behavior is similar to the execution time results, with the biggest reductions for BT, CG, LU, SP and UA. Additionally, we can observe that the DRAM energy was reduced more than the processor energy (11.3% and 5.3% on average, respectively) because a locality-aware mapping has more influence in the memory than in the processor. We also measured the energy per instruction, which was reduced by 4.4% on average, and up to 12.5% in SP. The results of energy per instruction show that our mechanism not only saves energy by reducing the execution time, but also by providing a more efficient execution, which is an important goal for future Exascale architectures [27].

4.3. Design Space Exploration

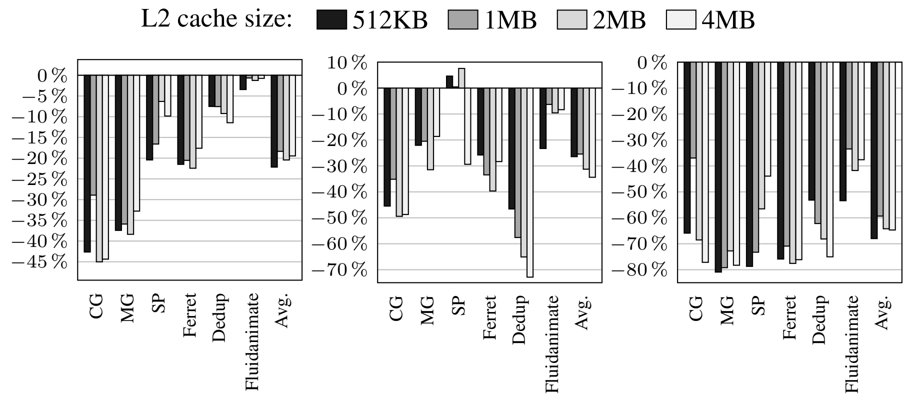
In this section, we analyze how LAPT behaves on environments with configuration different from the previous experiments using Simics/GEMS. We vary four important parameters: cache memory size, memory page size, interchip interconnection latency and the multiplication threshold. We show only the results regarding the applications whose communication patterns are illustrated in Figure 2 due to space limitations.

4.3.1. Varying Cache Sizes

We first analyze how LAPT behaves under different cache sizes. The cache memory size influences the performance improvements obtained with locality-aware mapping because it affects the amount of cached shared data and amount of accesses to the main memory. The previous experiments were performed with L2 cache memories with a capacity of 1 MByte and a latency of 5 cycles. We now show results of caches with 512 KBytes, 2 MBytes, and 4 MBytes, and latencies of 4, 6, and 6 cycles, respectively, calculated using CACTI [28]. We analyze execution time, cache misses and interchip interconnection. Figure 6 contains the improvements of LAPT over the baseline. In the absolute values, we chose to show cycles per instruction (CPI) instead of execution time due to the high difference of execution time between the applications, which could make the graphics illegible.

LAPT was able to improve performance for all applications and all cache sizes. The results follow the same pattern of the previous experiments: the amount of reduction of execution time depends on the reduction of cache misses and interchip interconnection traffic. In Dedup, LAPT presented the highest reduction of execution time with cache memories of 4 MBytes, in which LAPT also reduced cache misses and interchip traffic the most. The same happens in Ferret, but with a 2 MBytes cache. In SP, the configuration where LAPT presented the best improvements was with a 512 KBytes cache, where only the interchip traffic presented the highest reductions. In CG and MG, the configuration where LAPT presented the best improvements was with a 2 MBytes cache, but only cache misses presented the highest reductions.

We can make a similar analysis to verify the configurations in which LAPT performed worse. LAPT performed worse in CG and Fluidanimate when using 1 MByte caches, where the reduction of cache misses and interchip traffic was also the worst.



(a) Execution time. (b) L2 cache MPKI. (c) Interchip traffic.

Figure 6: Varying L2 cache memory sizes. Results are normalized to the default mapping with the corresponding cache size.

Dedup performs similarly, but with 512 KBytes. MG and SP performed worse with 4 MBytes and 2 MBytes caches, respectively, where only cache misses had the lowest reductions. LAPT was able to reduce only interchip traffic in SP with 512 KBytes, 1 MByte and 2 MBytes caches.

LAPT, in some situations, increases cache misses and thereby interchip traffic when increasing cache size. This happened mostly in CG and MG with 1 MByte caches, and in Ferret with a 4 MByte cache. In these applications, more thread migrations are introduced due to their communication patterns, in which a slight change may result in a different thread mapping. Depending on the migrations, more cache misses or interchip traffic can be introduced, decreasing performance. These thread migrations could be avoided if the thread mapping algorithm performed an analysis on the communication pattern to check if a migration is really necessary. Also, the memory addresses of pages change when they migrate to another node, which introduces cache misses.

It is important to note and reiterate that, despite the lower reduction in some configurations, LAPT was able to reduce execution time in all cases.

4.3.2. Varying Page Sizes

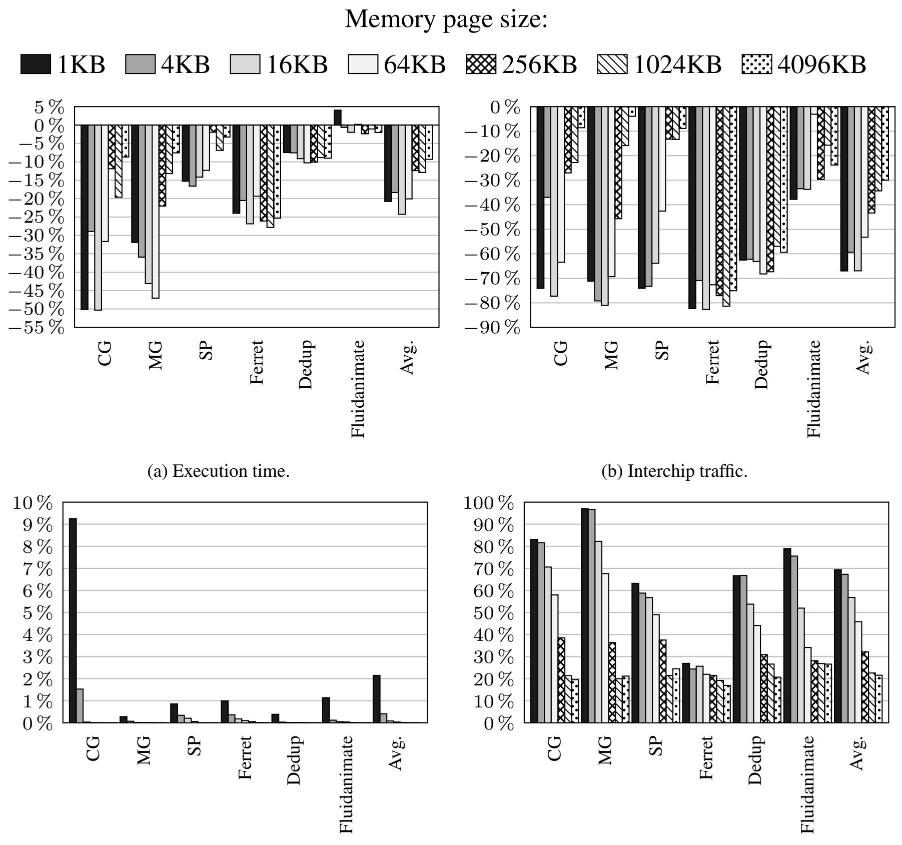
Another important parameter for LAPT is the page size. In this section, we evaluate how LAPT behaves under different page sizes. The normalized execution time and interchip interconnection traffic are shown in Figure 7a and 7b. We also show the TLB miss rate, in Figure 7c, and a metric called exclusivity level, in Figure 7d, introduced in [7]. The exclusivity level of a page corresponds to the highest number of accesses to the page from a single NUMA node in relation to the number of accesses from all nodes. The exclusivity level of an application is a weighted average of the exclusivity of all pages considering the amount of memory accesses. The higher the exclusivity level of an application, the higher its potential for locality-aware data mapping. The previous experiments were performed with a 4 KBytes page size. In this section we evaluate how LAPT performs with page sizes ranging from 1 KByte to 4 MBytes.

We can observe that, as expected, the TLB miss rate drops considerably when page size increases. This influences LAPT because there are less updates to the communication matrix, since the updates happen on TLB misses, influencing the detected communication pattern. Since LAPT detects communication in the page level granularity, increasing page size can also lead to the detection of different communication patterns. This happens when two threads access the same page, but in different offsets. The analysis of the exclusivity level shows that, for all applications except Ferret, this interference of accesses in different offsets is low when the page size is 1 KByte or 4 KBytes, but increases noticeably in larger page sizes.

In relation to data mapping, the lower exclusivity level with larger page sizes tends to decrease the performance improvement. The reason for this is that there would be more threads executing in cores from different NUMA nodes when the page size is larger. LAPT was able to decrease interchip traffic more with 1 KByte than with 4 MByte for all applications, showing the tendency of lower improvements with larger page sizes. In Ferret, this decrease is lower because the exclusivity level is less influenced by the page size than in the other applications. In CG, MG, SP and Fluidanimate, the exclusivity level is very similar between 1 MByte and 4 MByte pages, because the exclusivity is already very high with 1 MByte, such that there is no increase in the number of threads accessing each page with 4 MByte pages. In SP, the exclusivity is a little higher with 4 MByte pages due to noise from different simulation instances.

In most cases, the best performance improvements happened with an intermediate page size, such as in MG with 64 KBytes page size. This happened because the pages were migrated earlier during the execution, such that the benefits of the improved data mapping took effect also earlier. Still in MG, the exclusivity level decreases significantly when the page size is 256 KByte, which is reflected in the lower reduction of interchip traffic and thereby in performance.

The reason for this behavior is that there is a trade-off between the efficiency of the behavior detection and the performance benefits of an improved data mapping. Smaller pages require more time to detect the pattern for the migration, but allow higher performance improvements since the lower granularity reduces the number of cross-node memory accesses (higher exclusivity). On the other hand, detecting the access pattern of larger pages can be more efficient, since a larger memory area can be characterized



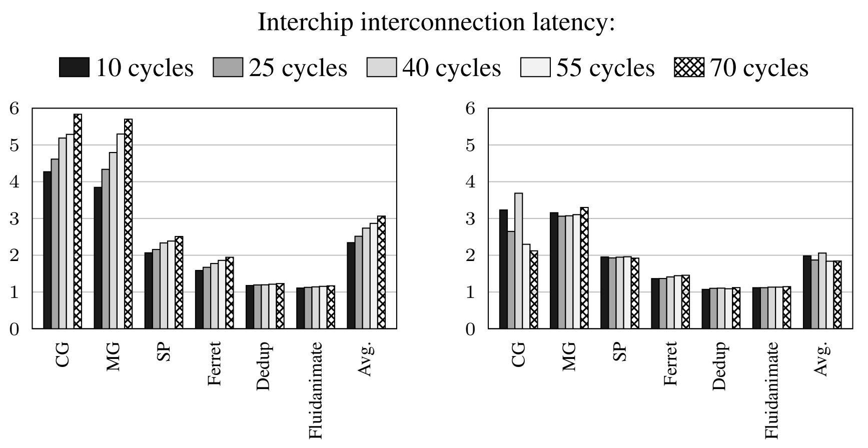
(a) Execution time.

(b) Interchip traffic.

(c) TLB miss rate.

(d) Exclusivity level.

Figure 7: Varying memory page sizes. Results of Figures 7a and 7b are normalized to the default mapping with the corresponding page size.



(a) Absolute CPI of OS. (b) Absolute CPI of LAPT.

Figure 8: Varying interchip interconnection latency.

with less TLB misses. Larger pages also cause more memory accesses across nodes, which limits the benefits of mapping.

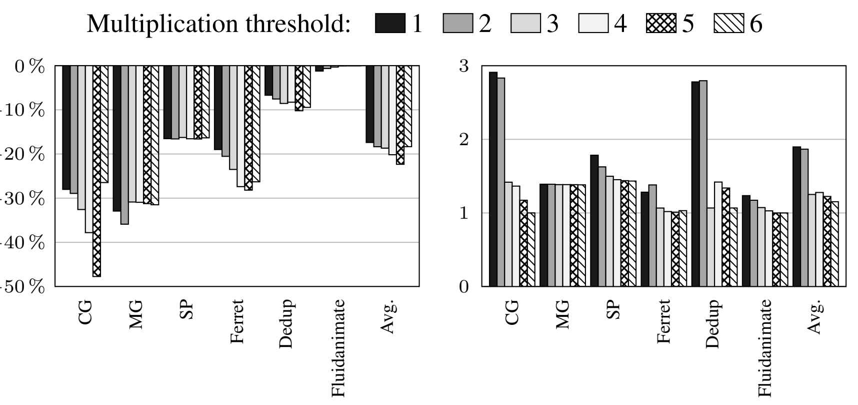
It is important to note that the memory footprint of the applications used in the simulations is very small due to the small input size used required by simulation time constraints. Applications with higher memory footprints keep similar exclusivity levels with larger page sizes [7]. Therefore, LAPT would keep similar performance improvements with larger page sizes.

4.3.3. Varying Interchip Interconnection Latencies

We also analyzed the interchip interconnection latency. It influences the time it takes to send cache coherence messages, such as cache line invalidations, as well as cache line transfers between caches, affecting thread mapping. Regarding data mapping, the interchip interconnection latency influences the time it takes to perform remote memory accesses. Local memory accesses do not suffer any impact. The previous experiments were performed with an interchip interconnection latency of 40 cycles. In this section we evaluate how LAPT behaves with latencies from 10 to 70 cycles.

The results obtained from varying the interchip interconnection latencies are shown in Figure 8. We can observe that the CPI of the execution of the operating system increases significantly with the increase of the interchip latency for most applications. On the other hand, for all applications except CG, the CPI obtained by LAPT suffers a much lower impact from latency increase. LAPT is less influenced by latency because it is able to reduce the amount of coherence messages, cache-to-cache transfers and remote memory accesses.

In CG, the decrease of CPI when increasing the latency happened due to the reduction of cache misses. This reduction of cache misses in CG occurred due to the fact that there were less thread migrations in the execution with a higher latency. As explained before, small differences in the communication pattern of CG from different executions can lead to different thread mappings, and thereby to different results. What is most



(a) Reduction of execution time. (b) Number of page migrations per page.

Figure 9: Varying the multiplication threshold. Figure 9a is normalized to the default mapping.

important is, despite these differences introduced by noise during execution, LAPT, as previously explained, suffers a lower impact from higher interconnection latencies.

4.3.4. Varying the Multiplication Threshold

In the previous experiments, we used Equation 6 to determine if a page should migrate to another node. The equation determines that a page migration will happen only if the value of the counter of the node with highest NV_P is greater or equal to twice the average value. In this section, we analyze the sensitivity of the mechanism to multiplication thresholds other than 2. We vary the threshold from 1 to 6 and evaluate the impact on execution time and number of page migrations. The execution time is shown in Figure 9a, and the number of migrations per page is shown in Figure 9b.

We can observe the tendency that, when increasing the multiplication factor, the number of page migrations decreases. This is the expected behavior, since the higher the multiplication factor in Equation 6, more accesses from a node are required to trigger a page migration. The applications most affected by the number of page migrations are CG and Ferret, where LAPT performs better with less page migrations. This happened not only due to a lower overhead of copying the pages across nodes, but also due to lowering its side effects, such as the cache misses and pollution (the address of the page changes when moving to another node) and interconnection traffic of transferring the page. In CG, the best improvements happen when the threshold is set to 5. When the threshold is set to 6, the performance decreases because of two reasons: (i) some important page migrations did not happen; (ii) pages take more time to migrate. In the other applications, the influence of the multiplication factor is very low.

5. Conclusions and Future Work

In this paper, we presented LAPT, a mechanism that adds a support in hardware to detect accurate memory access patterns between threads and information about which threads access each page. In the operating system, LAPT analyses the detected information and performs locality-aware thread and data mapping. In contrast to previous work, LAPT has access to many more memory access samples and does not require any previous information about the behavior of the applications, nor changes to the application or runtime libraries.

We evaluated LAPT in Simics/GEMS and on a real machine, achieving performance improvements of up to 35.9% and 19.2% (10.1% and 6.7% on average). Performance improvements were possible due to a reduction of cache misses and traffic on the interconnections. L3 cache MPKI and interconnection traffic were reduced by up to 59.8% and 97.9% in the real machine (18.2% and 41.5% on average). Energy consumption was reduced by up to 15.7% (5.3% on average).

In the experiments with Simics/GEMS, we varied several architectural parameters, evolving cache memory and memory page sizes, as well as different interchip interconnection latencies. Experiments with different cache sizes indicated that the importance of thread mapping increases when the cache memory size also increases, as there will be more shared data in the caches. The importance of data mapping is higher when the memory footprint of the application overwhelms the available cache memory space. Experiments with different page sizes showed that LAPT is able to handle page sizes higher than the default of x86 architectures. When varying interchip interconnection latencies, the impact of higher latencies is lower in LAPT due to the high reduction of accesses through interchip interconnections. Also, these experiments demonstrated that LAPT is able to improve performance in a wide range of architectures.

As future work, we intend to evaluate other thread mapping algorithms. We also plan to extend LAPT to support parallel applications with several processes that do not necessarily share a common page table.

Acknowledgment

This work was partially supported by CNPq, Capes and FAPERGS. It was developed in the context of the International Projects HPC-GA and HOSCAR.

References

[1] R. Azimi, D. K. Tam, L. Soares, M. Stumm, Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring, ACM SIGOPS Operating Systems Review 43 (2) (2009) 56–65. doi:10.1145/1531793.1531803.

[2] J. Marathe, V. Thakkar, F. Mueller, Feedback-Directed Page Placement for ccNUMA via Hardware-generated Memory Traces, Journal of Parallel and Distributed Computing 70 (12) (2010) 1204–1219.

[3] J. Marathe, F. Mueller, Hardware Profile-guided Automatic Page Placement for ccNUMA Systems, in: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2006, pp. 90–99.

[4] E. H. M. Cruz, M. Diener, M. A. Z. Alves, P. O. A. Navaux, Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols, Journal of Parallel and Distributed Computing 74 (3) (2014) 2215–2228. doi:10.1016/j.jpdc.2013.11.006.

[5] N. Barrow-Williams, C. Fensch, S. Moore, A Communication Characterisation of Splash-2 and Parsec, in: IEEE International Symposium on Workload Characterization (IISWC), 2009. doi:10.1109/IISWC.2009.5306792.

[6] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, R. Namyst, Structuring the execution of OpenMP applications for multicore architectures, in: IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2010.

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

[8] E. H. Cruz, M. Diener, M. A. Alves, L. L. Pilla, P. O. Navaux, Optimizing Memory Locality Using a Locality-Aware Page Table, in: International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2014, pp. 198–205. doi:10.1109/SBAC-PAD.2014.22.

[9] T. Klug, M. Ott, J. Weidendorfer, C. Trinitis, autopin – Automated Optimization of Thread-to-Core Pinning on Multicore Systems, High Performance Embedded Architectures and Compilers 3 (4) (2008) 219–235.

[10] P. Radojković, V. Cakarević, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, M. Valero, Thread Assignment of Multithreaded Network Applications in Multicore/Multithreaded Processors, IEEE Transactions on Parallel and Distributed Systems (TPDS) 24 (12) (2013) 2513–2525.

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

[12] S. Bokhari, On the Mapping Problem, IEEE Transactions on Computers C-30 (3) (1981) 207–214.

[13] F. Pellegrini, Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs, in: Scalable High-Performance Computing Conference (SHPCC), 1994, pp. 486–493.

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

[15] T. Hoefler, M. Snir, Generic topology mapping strategies for large-scale parallel architectures, in: International Conference on Supercomputing (ICS), 2011. doi:10.1145/1995896.1995909.

[16] G. Karypis, V. Kumar, A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs, SIAM Journal on Scientific Computing 20 (1) (1998) 359–392.

[17] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, U. V. Catalyurek, Parallel hypergraph partitioning for scientific computing, in: IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2006. doi:10.1109/IPDPS.2006.1639359.

[18] E. H. M. Cruz, M. Diener, L. L. Pilla, P. O. A. Navaux, An Efficient Algorithm for Communication-Based Task Mapping, in: International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2015.

[19] E. Jeannot, G. Mercier, F. Tessier, Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques, IEEE Transactions on Parallel and Distributed Systems 25 (4) (2014) 993–1002. doi:10.1109/TPDS.2013.104.

[20] Intel, 2nd Generation Intel Core Processor Family, Tech. Rep. September (2012).

[21] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, A. Tal, Analyzing Parallel Programs with Pin, IEEE Computer 43 (3) (2010) 34–41.

[22] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, Simics: A Full System Simulation Platform, IEEE Computer 35 (2) (2002) 50–58. doi:10.1109/2.982916.

[23] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, D. Wood, Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset, ACM SIGARCH Computer Architecture News 33 (4) (2005) 92–99.

[24] N. Agarwal, T. Krishna, L.-S. Peh, N. K. Jha, GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator, in: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2009, pp. 33–42. doi:10.1109/ISPASS.2009.4919636.

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

[26] C. Bienia, S. Kumar, J. P. Singh, K. Li, The PARSEC Benchmark Suite: Characterization and Architectural Implications, in: Parallel Architectures and Compilation Techniques (PACT), 2008, pp. 72–81.

[27] J. Torrellas, Architectures for extreme-scale computing, IEEE Computer 42 (11) (2009) 28–35.

[28] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, N. P. Jouppi, P. Alto, Cacti 5.1, Tech. rep. (2008).