2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing

Optimizing Memory Locality Using a Locality-Aware Page Table

Eduardo H. M. Cruz, Matthias Diener, Marco A. Z. Alves, Laércio L. Pilla, Philippe O. A. Navaux
Informatics Institute - Federal University of Rio Grande do Sul - Porto Alegre - Brazil
{ehmcruz, mdiener, mazalves, llpilla, navaux}@inf.ufrgs.br

***Abstract*—One of the main challenges for modern parallel shared-memory architectures are accesses to main memory. In current systems, the performance and energy efficiency of memory accesses depend on their locality: accesses to remote caches and NUMA nodes are more expensive than accesses to local ones. Increasing the locality requires knowledge about how the threads of a parallel application access memory pages. With this information, pages can be migrated to the NUMA nodes that access them (data mapping), as well as threads that access the same pages can be migrated to the same node such that locality can be improved even further (thread mapping).**

In this paper, we propose LAPT, a mechanism to store the memory access pattern of parallel applications in the page table, which is updated by the hardware during TLB misses. This information is used by the operating system to perform an optimized thread and data mapping during the execution of the parallel application. In contrast to previous work, LAPT does not require any previous information about the behavior of the applications, or changes to the application or runtime libraries. Extensive experiments with the NAS Parallel Benchmarks (NPB) and PARSEC showed performance and energy efficiency improvements of up to 19.2% and 15.7%, respectively, (6.7% and 5.3% on average).

I. Introduction

Multi-core processors, while providing an increased computational power, create a high pressure on the memory subsystem. To reduce the average latency per memory request, processors have a complex memory hierarchy, formed by multiple cache levels, some composed of multiple banks connected to a memory controller. Outside the processor chip, the memory controller interfaces a Uniform or Non-Uniform Memory Access (UMA or NUMA) DRAM system.

During the execution of a multi-threaded application, the placement of its threads and their data can have a great impact on performance and energy consumption, which are important goals for current and future architectures [31]. Considering that each processor family uses a different organization of the cache hierarchy and the main memory system, the impact of the execution time also varies among different systems. In general, the cache hierarchy is formed by multiple levels, where the levels closer to the processor cores tend to be private, followed by caches shared by multiple cores. For NUMA systems, besides the cache hierarchy, the main memory is also clustered between cores or processors.

Improving the data locality in NUMA systems can reduce the number of accesses to remote nodes as most of the threads' data will be kept close to them [24]. Moreover, the thread placement can reduce the number of cache invalidations and line replications, considering that threads that share data will be placed closer, sharing the same memory resources [2]. These thread and data mappings can impact different resources, such as interconnection systems and cache coherence protocols.

Several related mechanisms for thread and data mapping have been proposed. Most of the proposals perform thread or data mapping separately [2], [13], [23], [10]. Several approaches focus on static mapping using memory traces from previous executions in controlled environments such as simulators [4], which has a high overhead and is not able to handle dynamic behavior. Some mechanisms depend on particular APIs [7], or rely on indirect or incomplete information about the locality of memory accesses [2], [13].

In this paper, we propose a *Locality-Aware Page Table (LAPT)*, which enables operating systems to perform thread and data mapping. Our mechanism detects the locality of memory accesses in hardware, and performs the thread and data mappings in software. LAPT has the following main features:

- LAPT operates during the execution of the application, allowing the mapping to be performed dynamically, without needing previous information about the application's behavior.
- By detecting the locality in hardware, LAPT has access to many more memory access samples than previous mechanisms, generating more accurate information. More specifically, it can track all memory accesses that result in TLB misses.
- Any shared memory-based parallel application is able to benefit from the optimized mapping without modification to its source code.
- It does not depend on any particular parallelization API. Also, no special API support is required, since the mechanism can be implemented in the operating system.

To our knowledge, this is the first mechanism to perform both thread and data mapping that is independent from the parallelization API. The simulation results executing applications from the NAS and PARSEC benchmark suites show average performance gains of 10.6%. Experiments on a real machine showed average performance gains of 6.7%.

II. Related Work

Several related thread and data mapping techniques have been proposed. In [4], a method to statically detect the communication pattern for thread mapping has been proposed. The authors generate memory access traces in a simulator

1550-6533/14 $31.00 © 2014 IEEE
DOI 10.1109/SBAC-PAD.2014.22

and analyze them to generate a communication pattern. This method can provide a high accuracy, as it has access to information of the entire execution of the application. A similar mechanism was proposed in [24], [14]. To generate information to guide data mapping, they used features from the performance monitoring unit (PMU) of Itanium-2 to collect samples of the memory addresses accessed by each thread. The main drawbacks of these methods are that they require the analysis of traces, which is an expensive procedure, and are not able to handle changes in the application behavior. LAPT does not have these disadvantages, as it performs mapping dynamically.

The PMU of Itanium-2 is also used in [23], where the authors perform the data mapping dynamically. Their profiling mechanism requires traps to the operating system on every high latency memory load operation or TLB miss, imposing a high overhead. Therefore, the authors enable the profiling mechanism just during the beginning of each application, losing the opportunity to handle changes during the execution of the application. A similar approach is described in [30], but using hardware monitors on Ultrasparc-III platforms. Both mechanisms perform only data mapping, and do not consider the communication between the threads, which is important to improve the usage of cache memories and interconnections.

Other mechanisms that perform mapping dynamically depend on indirect statistics from hardware counters, such as instruction per cycle (IPC). One example is Autopin [21], which evaluates several thread mappings given as input and selects the one with the highest IPC. Similarly, Blackbox [27] selects the best mapping among 1000 random mappings. As these mechanisms have to evaluate several mappings, a lot of time is spent to find the best one. The probability of selecting the best possible mapping is very low, since the number of different mappings is exponential to the number of threads. Furthermore, the hardware counters they use are not accurate enough to represent communication. On the other hand, LAPT has direct access to the memory addresses accessed by each thread. ForestGOMP [7] also uses indirect statistics from hardware counters, and supports only OpenMP based applications, while LAPT is able to handle any shared memory based application.

In [2], the authors use hardware counters that provide memory addresses of requests solved by remote cache memories. It detects incomplete communication patterns, since memory requests resolved by local caches are not considered. SPCD [13] uses page faults of parallel applications to detect communication. Although these proposals have access to the memory addresses, only a small sample of all accesses is taken into account. LAPT guarantees that all memory pages are considered when detecting the locality by tracking the TLB misses. Furthermore, these previous proposals do not perform data mapping, and therefore are not able to reduce remote memory accesses in NUMA architectures.

III. LAPT – A Locality-Aware Page Table

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 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 page table in the main memory, a special cache memory, called the 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 to a physical address 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.

We introduce a *Locality-Aware Page Table* (LAPT), a page table with special fields that allows operating systems to perform thread and data mapping in parallel applications. LAPT is implemented on both hardware and software levels. On the hardware level, LAPT keeps track of all TLB misses, detecting the communication between the threads and registering a list of the latest threads that accessed each page 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 the pages to NUMA nodes by checking which threads accessed each page.

We first explain how we detect the communication in hardware. Afterwards, we describe how we use the detected communication to map threads and data. Finally, we discuss the overhead of our proposal.

A. Detecting the Communication

To identify the threads that access each page, LAPT requires the addition of an entry in the page table. We call this entry *communication vector* (CV). Each CV stores the IDs of the last threads that accessed the corresponding page. Whenever CV gets full, old thread IDs need to be removed to make room for the new ones, keeping temporal locality. LAPT also introduces a *communication matrix* (CM) in main memory for each parallel application, which stores an estimation of the amount of communication between each pair of threads. Special registers containing the memory address

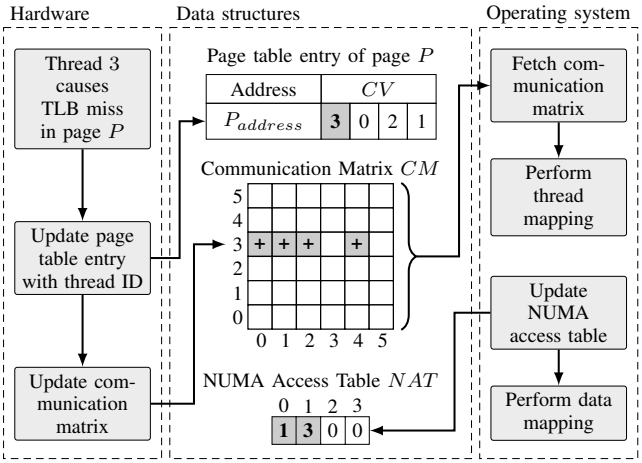


Fig. 1: Overview of the LAPT mechanism.

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 page 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:

$$CM[T][\ CV_P[i]\ ] \leftarrow CM[T][\ CV_P[i]\ ] + 1, \quad 0 \leq i < sizeof(CV_P) \tag{1}$$

After updating CM, LAPT inserts thread T into CV_P:

$$CV_P[i] \leftarrow CV_P[i-1], \qquad 0 < i < sizeof(CV_P) \tag{2}$$

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

B. Calculating the Thread Mapping

The information provided by the communication detection is used to calculate an optimized mapping from threads to processing units (PUs) during the execution of the parallel application. The mapping problem is NP-hard [6], therefore it requires the use of efficient heuristics to calculate the mapping. Dynamic mapping requires algorithms with a short execution time, since its overhead directly impacts the executing application.

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

To calculate the mapping, we use the dual recursive bipartitioning algorithm of the Scotch mapping library [26], version 6.0. We selected this algorithm because it has a short execution time (less than 1 ms to map 32 threads) while providing good results [28], and is therefore 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 algorithms, such as METIS [20], Zoltan [12] or TreeMatch [17], 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 the 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 dynamically adjust the frequency. If the calculated mapping does not differ from the previous mapping, we increase the mapping interval by 50 ms, since the mapping is stable. If the mapping differs, we divide the interval by 2. 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.

C. Calculating the Data Mapping

To calculate the data mapping, the operating system iterates over the page table of the application, verifying the contents of the communication vector CV of each page. In the operating system, we also keep a separate data structure to store an estimation of the amount of memory accesses from each NUMA node. We call this data structure *NUMA access table* (*NAT*). There is one entry in the *NAT* for each page. Each entry has one counter per NUMA node.

When the data mapping routine is called, for every page P of the application, the *NAT* of page P is incremented in the NUMA nodes of each thread of the CV of the corresponding page:

$$\mathit{NAT}[P][\ node(CV_P[i])\ ] \leftarrow \mathit{NAT}[P][\ node(CV_P[i])\ ] + 1, \quad 0 \leq i < sizeof(CV_P) \tag{4}$$

Where $node(x)$ is a function that returns the NUMA node in which thread x is running. After the contents of *NAT[P]* are updated, we use the following equations to determine if page P should be migrated to node N.

$$DataMap(P) = \{N \mid \mathit{NAT}[P][N] = max(\mathit{NAT}[P])\} \tag{5}$$

$$Migrate = \begin{cases} true & \text{if } max(\mathit{NAT}[P]) \geq 2 \cdot avg(\mathit{NAT}[P]) \\ false & \text{otherwise} \end{cases} \tag{6}$$

A page migration will happen only if the value of the counter of the node returned by $DataMap$ is greater or equal twice its average value. In this way, we reduce the possibility of having a ping-pong effect of page migrations between the 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 of *NAT[P]* is at least 1, we use an aging technique in which all elements of *NAT* are divided by 2, making it possible to adapt to changes in access behavior.

The operating system calls the data mapping routine in two situations. First, whenever the thread mapping routine is called and causes a change in the mapping. This is done in order to move the pages used by the threads when they migrate. Second, whenever there has been a long time since the last call, since the data mapping may change even if the thread mapping keeps the same. In our experiments, the maximum interval to call the data mapping routine was 500 ms.

D. Example of the Operation of LAPT

To illustrate how LAPT works, consider the example shown in Fig. 1, where an application that consists of 6 threads is executing on a NUMA machine with 4 nodes. The communication vector supports up to 4 thread IDs. Thread 3 tries to access page P, which 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 the execution, the operating system evaluates the communication matrix to update the thread mapping. The operating system 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 Access Table for data mapping. The initial value of all the elements in the *NAT[P]* is 0. Since only thread 0 is executing on node 0, the corresponding entry in the *NAT[P]* will be incremented by 1. Likewise, the other three threads that are in the CV are running on node 1, whose value will be incremented by 3. The node with highest element in *NAT[P]* is node 1, with value 3, and the average of the values of *NAT[P]* is 1. Therefore, page P is migrated to node 1 following Eq. 6.

E. Overhead

The overhead consists of storage space in the main memory, circuit area to implement LAPT in the processor, and runtime overhead.

1) Memory Storage Overhead: The following structures are stored in the main memory:

Communication matrix (CM): It 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 during the execution the parallel application creates more threads than the maximum supported by the communication matrix, the operating system can allocate a larger matrix and copy the values from the old matrix.

Communication vector (CV): It is stored on the entries of the last level page table. A common size for each page table entry in current architectures is 8 Bytes, as in x86-64 [16]. We extend the size of each entry to 16 Bytes, reserving 2 Bytes to store each thread ID. This allows us to track 4 threads per page, supporting up to 65536 threads per parallel application. The space required by CV represents less than 0.2% of memory space overhead compared to the original system when using a standard 4 KByte page size.

NUMA access table (NAT): We store one entry in the *NAT* per page the application uses. A multilevel hash is applied to index the *NAT* without collisions. We use 1 Byte to store each counter, which results in an extra 0.2% of storage overhead.

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 thread ID, 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 up. 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.

TABLE I: Configuration of the experiments.

	Parameter	Value
Real	Processors	2x Xeon E5-2650 (SandyBridge), 8 cores, 2-SMT
	Caches/processor	8x 32 KByte L1, 8x 256 KByte L2, 20 MByte L3
	System memory	32 GByte DDR3-1333, 4 KByte Page size
Simics	Processors	4x 2 cores processors, 2.0 GHz, 32 nm
	L1 cache/processor	2x16 KByte, 4-way, 1 bank, 2 cycles latency
	L2 cache/processor	1 MByte, 8-way, 2 banks, 5 cycles latency
	TLB/processor	2x TLBs, 64 entries, 4-way
	Cache coherence	directory-based MOESI protocol, 64 Byte line size
	Memory	8 GByte DDR3-1333 9-9-9, 4 KByte Page size
	Interconnection	1/40 cycles latency (intra/interchip) 64/16 Byte bandwidth (intra/interchip)

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. On the software level, the time overhead includes the calculation of the thread mapping and data mapping, and the migrations. An evaluation of the time overhead is found in Section V-C.

IV. Experimental Methodology

In this section, we describe how we evaluated our proposed mechanism. We describe the types of experiments we performed: evaluation on a full system simulator, and evaluation on a real machine. We also show which benchmarks we used as workload. Table I summarizes the parameters of the real and simulated machines.

A. Evaluation on a Full System Simulator

We implemented LAPT in the Simics full system simulator [22], extended with the GEMS-Ruby memory model [25] and the Garnet interconnection model [1]. The simulated machine runs the Linux kernel and consists of 4 processors, each with 2 cores, with private L1 caches, and L2 caches shared by all cores. Each processor is on a different NUMA node. Cache latencies were calculated using CACTI [29], and the memory timings were taken from JEDEC [18]. We simulate the benchmarks with small input sizes due to simulation time constraints. To compensate for the small input sizes, we reduced the size of caches memories and TLBs accordingly, as done in [11]. 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. we also used the Scotch library [26] in the oracle mapping, otherwise it would be unfeasible to calculate the thread mappings. The results are normalized to the default mapping.

B. Evaluation on a Real Machine

Since LAPT is an extension to current hardware, we simulate its behavior with Pin [3], a binary instrumentation tool. We used Pin for the analysis because it is faster than a full system simulator. To make it possible to evaluate LAPT in real machines, the information about the thread and data mappings generated inside our Pin tool are fed into the mapping mechanism during the execution of the applications.

The experiments in the real machine were performed in a two NUMA node machine, which has one Intel Xeon E5-2650 processor per node, with a total of 32 virtual cores. The machine runs version 3.8 of the Linux kernel. Information about the hardware topology was gathered automatically using Hwloc [8]. Besides performance, we measured L2 and L3 cache misses per thousand instructions (MPKI), as well as energy consumption using the Running Average Power Limit (RAPL) hardware counters [16] with Intel PCM. Results regarding the traffic on the QuickPath Interconnect (QPI) were estimated using Intel VTune by measuring the amount of cache to cache transfers and remote memory accesses.

All experiments in the real machine were executed 30 times. We show average values as well as a 95% confidence interval calculated with Student's t-distribution. 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. The operating system mapping is the original scheduler and Autonuma [9] data mapping policy of the Linux kernel. It represents the baseline for our experiments. For the random mapping, we randomly generated a thread and data mapping for each execution. 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 [4], and use the Scotch library [26] to calculate the thread mappings. We disabled the automatic kernel thread scheduling and Autonuma for LAPT, and for the random and oracle mappings to avoid interference. All results are normalized to the average of the operating system mapping.

C. Workloads

As workloads, we used the OpenMP implementation of the NAS Parallel Benchmarks (NPB) [19], v3.3.1, and the PARSEC benchmark suite [5], v3.0. For the evaluation in the real machine, the evaluated application must present the same memory address space across different executions. This is required because the trace generated in Pin is used to guide mapping decisions. For this reason, only the NAS applications, except DC, were executed in the real machine. We executed all NAS applications with one thread per virtual core (32 threads in total). The number of threads for PARSEC applications was also set to 32 threads, but the actual number of created threads vary for each application.

Input sizes were chosen to provide similar total execution times and feasible simulation time. Regarding NAS, benchmarks BT, LU, SP and UA were executed using input size *A* in the real machine, and input size *W* in Simics. Benchmarks CG, EP, FT, IS and MG were executed using input size *B* in the real machine, and input size *A* in Simics. DC was executed with input size *W* in Simics. Regarding PARSEC, the input size used in Canneal was *simmedium*, and all other applications used *simlarge*.

V. Experiments and Results

In this section, we evaluate the performance and energy consumption improvements of our proposal, as well as its overhead.

A. Performance Results

The results obtained in Simics and real machine can be found in Fig. 2 and 3, respectively. The communication patterns of a subset of our workloads are depicted in Fig. 4. Since the absolute values of the contents of the communication matrices vary significantly between the 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 between threads.

Some applications are influenced by both thread and data mappings. One example of such an application is BT. The communication pattern of BT, shown in Fig. 4a, highlights that there are threads that communicate more with a small subgroup of threads. When an application presents this characteristic, mapping threads that communicate to nearby cores in the memory hierarchy improves performance.

In the case of BT, most communication happens between neighboring threads, which is very common on domain decomposition parallel applications. Other communication patterns also are suitable for mapping. For instance, the communication between more distant threads in MG is more evident than in the other applications. In Ferret, nearby threads form clusters. Optimized mappings usually reduce cache misses and interconnection traffic, as observed in BT. In MG, only interconnection traffic was reduced.

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 communication pattern of Vips, shown in Fig. 4d, where each pair of threads has a similar amount of communication. On the other hand, the

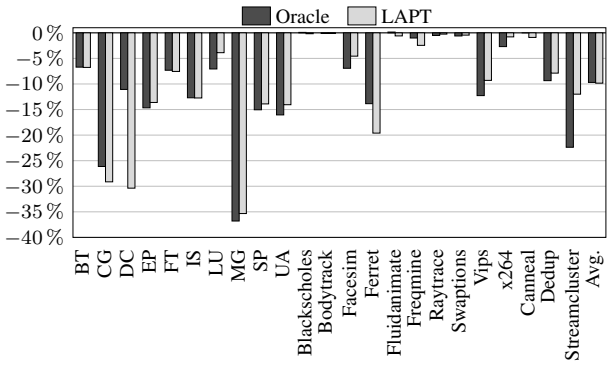
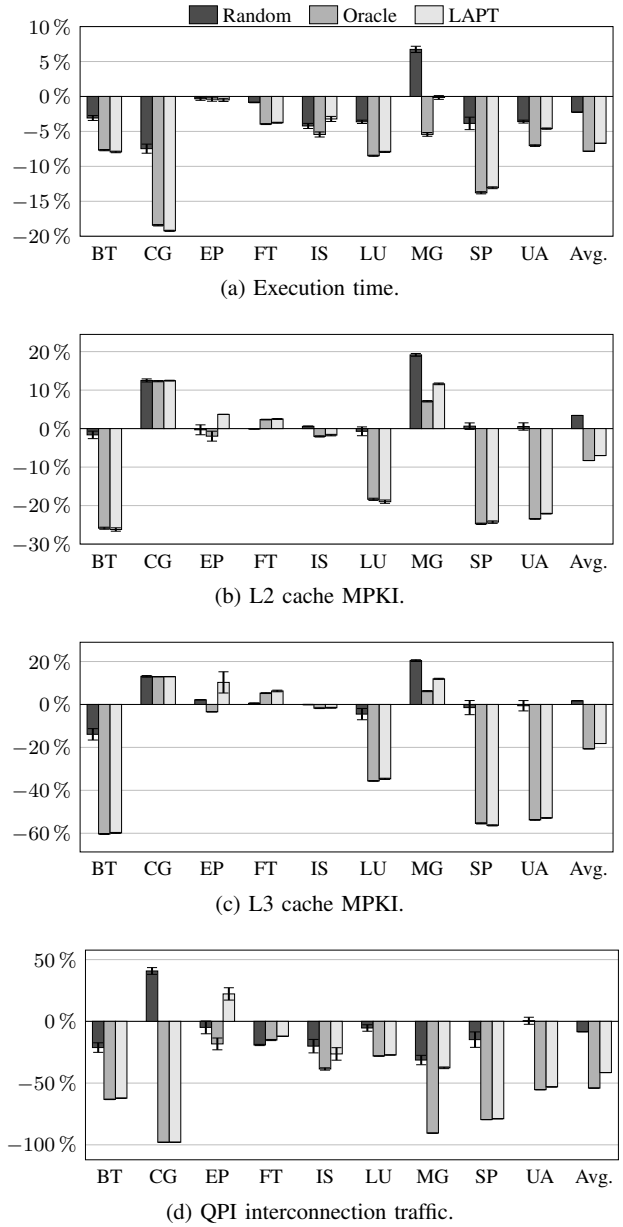


Fig. 2: Execution time in Simics, normalized to the default mapping.



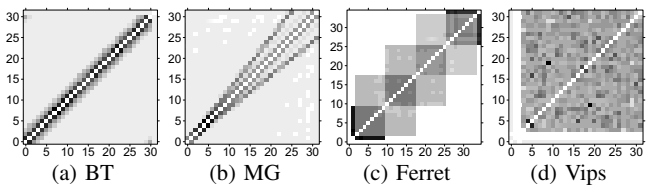
(a) Execution time.

(b) L2 cache MPKI.

(c) L3 cache MPKI.

(d) QPI interconnection traffic.

Fig. 3: Performance results on the real machine, normalized to the OS.



(a) BT (b) MG (c) Ferret (d) Vips

Fig. 4: Communication patterns that represent several applications. Axes represent thread IDs. Cells show the amount of accesses to shared pages for each pair of threads. Darker cells indicate more accesses.

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.

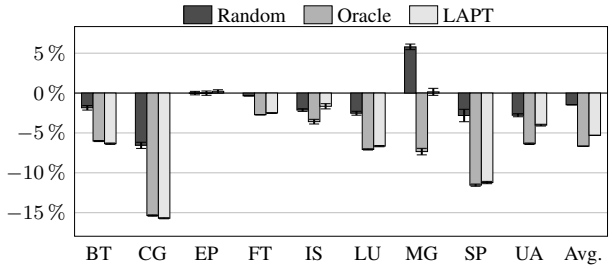
The communication pattern of Swaptions is similar to the one of Vips, not being influenced by thread mapping. Also, the memory usage of Swaptions is low, such that almost all its data fits into the caches and are thereby not affected by data mapping. EP is a CPU-bound application [19] with almost no 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 the performance.

LAPT reduced the L2 and L3 MPKI in the real machine by 7.0% and 18.2% on average. L3 MPKI had a higher reduction than L2 MPKI due to the higher cache size, which allows more caching of shared data. The interconnection traffic was reduced by an average of 41.5%. Execution time was reduced by 6.7% and 10.6% on average in the real machine and Simics, respectively. The execution time impact tends to be a fraction of the cache miss and the interconnection traffic improvements, because the execution time is influenced by several other factors besides the data accesses. Therefore, a smaller reduction of execution time compared to cache misses and interconnection traffic is expected.

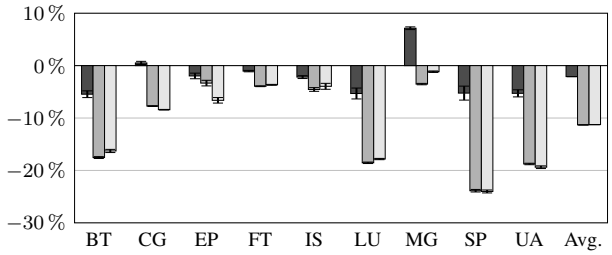
The results obtained with LAPT are similar to the oracle mapping, demonstrating its effectiveness. Occasionally, LAPT performed better than the oracle mapping. This may happen because we only consider the memory accesses to generate the oracle mapping, while there are several other factors that influence the 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 migrations introduced by the operating system, but due to a more efficient usage of the machine resources.

B. Energy Consumption Results

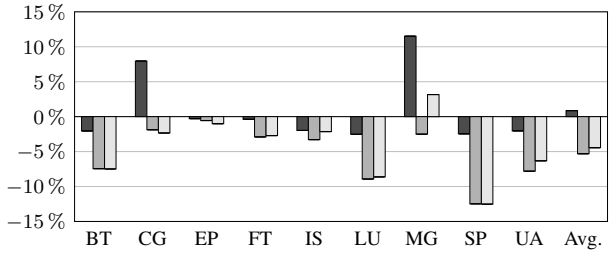
Results of the total amount of processor and DRAM energy consumption are shown in Fig. 5a and 5b. We can observe that the behavior is similar to the execution time results, with the biggest reductions for BT, CG, LU, SP and UA. The other applications show no difference or a small reduction of energy consumption. 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 communication-aware mapping has more influence on the memory than on the processor. The results of energy per instruction, in Fig. 5c, show that our mechanism not only saves energy by reducing the executing time, but also by providing a more efficient execution, which is an important goal for future exascale architectures [31]. Energy per instruction was reduced by 4.4% on average, and up to 12.5%, in SP.



(a) Package energy.



(b) DRAM energy.



(c) Total energy per instruction.

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

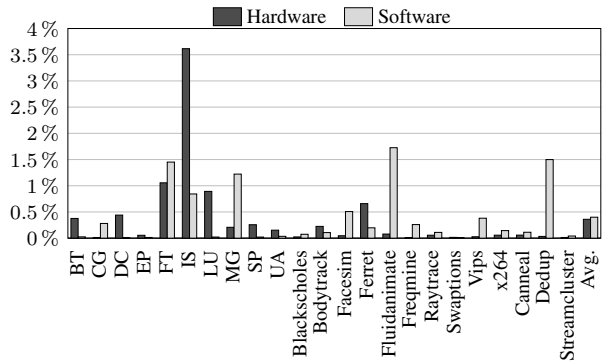


Fig. 6: Performance overhead on the hardware and software levels in Simics.

C. Performance Overhead

As explained in Section III-E3, LAPT introduces a performance overhead on the hardware and software levels. We show the overhead only for the machine simulated in Simics because LAPT is not present in real machines. In the hardware level, IS presented a high performance overhead because it has a high TLB miss rate, introducing many memory accesses to update the page table entries and communication matrix. The applications FT, MG, Fluidanimate and Dedup present high software overhead because they migrate a lot of pages in relation to their execution time. The average performance overhead on the hardware and software levels was only 0.36% and 0.4%, respectively.

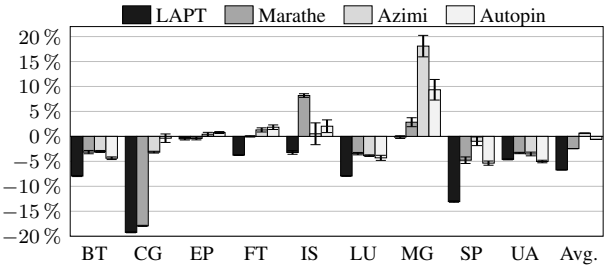


Fig. 7: Performance comparison to related work in the real machine, normalized to the OS.

D. Comparison to Related Work

We compare LAPT to three previous techniques: Autopin [21], the Azimi thread mapping [2], and to the Marathe [23] data mapping mechanism. Autopin was executed with 5 mappings: the Oracle mapping, as well as 4 random mappings. After a warmup time of 500 ms, every mapping was evaluated for 150 ms. Then, the mapping that resulted in the highest IPC was selected for the rest of the execution. Autopin was directly executed on the real machine. We implemented Azimi and Marathe in Pin, generating mapping information that is fed to a runtime system during the execution of the application in the real machine, as in Section IV-B. For Azimi, the system simulated inside Pin consisted of 2 level caches, each cache with 16 MByte, 16-way set associative, and a MOESI cache coherence protocol. For Marathe, we used the same cache configuration as for Azimi, and a long latency load based profile, as described in [23].

Figure 7 shows the execution time of LAPT and the related mechanisms. Values are normalized to the results of the operating system. In CG, Marathe presented results similar to LAPT. This happens because, as previously explained, CG is an application only affected by data mapping. In applications such as BT and SP, Marathe performed much worse due to the lack of thread mapping. Autopin, in several executions, was not able to select the Oracle mapping. Even when it correctly selects the Oracle mapping, its performance improvement is lower than ours because it needs to evaluate several other mappings. Regarding Azimi, the incomplete communication pattern that it detected resulted in sub-optimal mappings. 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.

VI. Conclusions and Future Work

The memory hierarchies of current hardware architectures impose different communication and memory access latencies, which are important to be considered when mapping threads to cores and pages to NUMA nodes, influencing the performance and energy consumption. However, current architectures do not provide accurate information to guide locality-aware thread and data mapping policies. In this context, we proposed LAPT, a mechanism that detects in hardware accurate communication patterns between threads, and knowledge about which threads access each page. On the software level, LAPT analyses the detected information and performs locality-aware thread and data mapping. LAPT represents an improvement to the state-of-art because it performs both thread and data mapping dynamically, and is based on accurate information about the memory access behavior.

We evaluated LAPT in Simics/GEMS and on a real machine, achieving performance improvements of up to 35.4% and 19.2% (10.6% and 6.7% on average). The 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 59.8% and 97.9% (18.2% and 41.5% on average). Energy consumption was reduced by up to 15.7% (5.3% on average). The extra circuit area to implement LAPT in current architectures represents less than 0.009% of the total area. The average performance overhead on the hardware and software levels was only 0.36% and 0.4%, respectively.

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.

References

[1] N. Agarwal, T. Krishna, L.-S. Peh, and 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)*, Apr. 2009, pp. 33–42.

[2] R. Azimi, D. K. Tam, L. Soares, and M. Stumm, "Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 56–65, Apr. 2009.

[3] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal, "Analyzing Parallel Programs with Pin," *IEEE Computer*, vol. 43, no. 3, pp. 34–41, 2010.

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

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[6] S. Bokhari, "On the Mapping Problem," *IEEE Transactions on Computers*, vol. C-30, no. 3, pp. 207–214, 1981.

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

[8] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications," in *Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010, pp. 180–186.

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

[10] E. H. M. Cruz, M. Diener, M. A. Z. Alves, and P. O. A. Navaux, "Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 74, no. 3, pp. 2215–2228, 2014.

[11] B. Cuesta, A. Ros, M. E. Gomez, A. Robles, and J. Duato, "Increasing the Effectiveness of Directory Caches by Avoiding the Tracking of Non-Coherent Memory Blocks," *IEEE Transactions on Computers*, vol. 62, no. 3, pp. 482–495, 2013.

[12] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, "Parallel hypergraph partitioning for scientific computing," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2006.

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

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

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

[16] Intel, "2nd Generation Intel Core Processor Family," Tech. Rep., 2012.

[17] E. Jeannot and G. Mercier, "Near-optimal placement of MPI processes on hierarchical NUMA architectures," in *Euro-Par*, 2010.

[18] JEDEC, "DDR3 SDRAM Standard," 2012.

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

[20] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, Jan. 1998.

[21] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis, "autopin – Automated Optimization of Thread-to-Core Pinning on Multicore Systems," *High Performance Embedded Architectures and Compilers*, vol. 3, no. 4, pp. 219–235, 2008.

[22] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, 2002.

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

[24] J. Marathe, V. Thakkar, and F. Mueller, "Feedback-Directed Page Placement for ccNUMA via Hardware-generated Memory Traces," *Journal of Parallel and Distributed Computing*, vol. 70, no. 12, pp. 1204–1219, 2010.

[25] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

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

[27] P. Radojković, V. Cakarević, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Thread Assignment of Multithreaded Network Applications in Multicore/Multithreaded Processors," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 24, no. 12, pp. 2513–2525, 2013.

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

[29] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, N. P. Jouppi, and P. Alto, "Cacti 5.1," Tech. Rep., 2008.

[30] M. M. Tikir and J. K. Hollingsworth, "Hardware monitors for dynamic page migration," *Journal of Parallel and Distributed Computing*, vol. 68, no. 9, pp. 1186–1200, Sep. 2008.

[31] J. Torrellas, "Architectures for extreme-scale computing," *IEEE Computer*, vol. 42, no. 11, pp. 28–35, 2009.