CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2014; **00**:1–23
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Communication-Aware Thread Mapping Using the Translation Lookaside Buffer

Eduardo H. M. Cruz*, Matthias Diener, Philippe O. A. Navaux

Informatics Institute, Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil

SUMMARY

Threads of parallel applications need to communicate in order to fulfill their tasks. The communication performance between the cores in modern multi-core architectures differs due to the memory and interconnection hierarchies. In these architectures, it is important to map the threads of parallel applications by taking into account the communication between them, to improve their performance and energy consumption. In parallel applications based on shared memory, communication is implicit, which makes it difficult to detect the communication pattern between the threads.
In this paper, we introduce a new light-weight mechanism to detect the communication pattern between threads of shared memory applications using the Translation Lookaside Buffer (TLB). Our mechanism relies on hardware features, which makes it transparent to the programmer and allows the detection to be performed by the operating system during the execution of the application. We also developed a heuristic mapping algorithm that use the detected pattern to dynamically map the threads to cores. Experiments were performed with applications from the NAS-OMP and PARSEC parallel benchmark suites in a simulated as well as a real machine. Results show that our mechanism can substantially improve parallel application performance, as well as processor and DRAM energy consumption.

Received . . .

KEY WORDS: Thread mapping; Shared memory; Translation Lookaside Buffer;

1. INTRODUCTION

As we approach the limits of instruction level parallelism, processor architectures are becoming more dependent on thread level parallelism to keep on increasing the performance, employing several cores to compute in parallel. In these multi-core architectures, threads of parallel applications need to communicate through the memory hierarchy by accessing shared data. These memory accesses have a substantial impact on the performance and the energy consumption, which are key factors for current and future computer system [1, 2]. Furthermore, communication performance between cores of multi-core architectures varies, since resources such as cache memories and interconnections may be shared by only a subgroup of cores. For these reasons, communication is an important factor to be considered when mapping threads to cores [3], such that threads that communicate execute on cores that are close to each other in the memory hierarchy.

This *communication-aware* thread mapping provides several benefits, such as reducing the number of cache misses and traffic on the interconnections. The reduction of cache misses occurs because the shared data tend to be stored in a cache shared by the communicating threads. In this way, less cache misses are expected, since some cache lines would store data shared by more than one thread, and a thread could access data that has already been accessed by another

*Correspondence to: Informatics Institute, Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil

thread. Likewise, the space overhead of replicated cache lines is reduced. Decreasing the number of replicated cache lines optimizes the usage of the caches [4] and also reduces the amount of invalidation messages sent by cache coherence protocols. Furthermore, mapping threads according to the communication can reduce energy consumption, since each coherence and cache line transfer message between the caches increases the amount of energy used by the interconnections.

Communication-aware mapping requires a method to detect the communication between the threads and algorithms that use this information to perform the mapping of threads to cores. In shared memory architectures, detecting the communication between threads presents challenges, because the communication is implicit and happens through memory accesses to shared variables. Most previous research in this area focuses on static profiling methods to provide information for the mapping [5, 6], which usually present a high overhead and can not be used in case the application's behavior changes between executions. Dynamic methods were also proposed, but with several disadvantages, such as low accuracy [7, 8], the need to modify the source code of the applications [9] or parallelization libraries, or they are limited to specific processor architectures [10].

In this paper, we propose a new light-weight, dynamic mechanism to detect the communication patterns of parallel applications based on shared memory. Our approach consists of looking at the most recently accessed virtual memory pages by each core. This is done by checking the contents of the *Translation Lookaside Buffer (TLB)*, which performs the translation of virtual addresses to physical addresses and is present in most architectures that support virtual memory. As there is one TLB for each core, the communication pattern can be detected by searching the TLBs for matching entries. Architectures with multi-threaded cores (such as SMT) are also supported, because the TLBs keep information about which virtual core accessed each entry. The detected communication pattern is then used to map the threads to cores using a heuristic algorithm based on Edmonds' graph matching algorithm.

This paper is an extension of our previous work [11], where we introduced the basic concept of communication detection with the help of the TLB and applied it in a static mapping context. The main extensions and improvements in this paper are the following. The communication detection and mapping is performed completely dynamically, during the execution of the parallel application. We improved the detection mechanism to unify the approaches for software-managed and hardware-managed TLBs. The improved detection mechanism has a higher accuracy and adapts better to the behavior of each parallel application due its trigger based on TLB misses, with a better support to detect dynamic behavior. It also presents a lower overhead by evaluating the TLBs in parallel, and scales better because its overhead is less dependent on the number of threads. We also improved the mapping algorithm to support a wider range of parallel architectures, and to use a communication filter to reduce the overhead of mapping calculations and unnecessary thread migrations. The evaluation of the mechanism was also improved by performing experiments with more benchmarks and measurement of the energy consumption.

Our mechanism has several advantages compared to existing approaches. The overhead on the executed application is low, since it does not require expensive operations such as simulation or memory tracing. It detects communication dynamically during the execution of the application, without the need for prior knowledge about the behavior of the application. The communication behavior is measured directly from the operation of the application, and not estimated using indirect measures, such as cache statistics. Finally, no modifications to the source code of the application or support libraries are required and it is independent of the parallelization API.

This paper is organized as follows. Section 2 describes how communication-aware thread mapping improves performance and energy consumption. Section 3 discusses related work. Section 4 explains our communication detection mechanism. Section 5 details our thread mapping algorithm. Section 6 shows our experiments. Finally, Section 7 draws our conclusions.

DOI: 10.1002/cpe

2. IMPROVING PERFORMANCE AND ENERGY CONSUMPTION THROUGH COMMUNICATION-AWARE THREAD MAPPING

To illustrate the potential for performance and energy improvements of communication-aware thread mapping, consider a common situation in shared memory programs, in which one thread writes to an area of memory and another reads from the same area. This is a producer-consumer behavior. If the cache coherence protocol is based on invalidation, such as *MESI* or *MOESI*, and the producer and consumer do not share a cache, an invalidation message is sent to the consumer every time the producer writes the data. As a result, the consumer always receives a cache miss when reading the data and needs to retrieve it from the cache of the producer, thereby causing more cache coherence traffic on the interconnections.

We implemented this producer-consumer benchmark, consisting of pairs of producer-consumer threads that communicate through a shared vector. The communication pattern between the threads changes periodically between two phases. These phases are depicted in Figure 1. In the first phase, shown in Figure 1a, the communication happens between neighboring threads. In the second phase, shown in Figure 1b, more distant threads are communicating. With this behavior, a static mapping technique is not able to maximize the performance, since the best mapping changes during the execution.

To evaluate the influence of communication-aware mapping, we executed this benchmark using three different mappings: the default mapping by the Linux OS scheduler, a static mapping based on the communication pattern of the first phase, and an oracle mapping, which migrates the threads at the beginning of each phase to the best mapping for that phase.

Figure 2 depicts the system architecture used in the experiment. It is based on the Intel SandyBridge microarchitecture [12]. In this architecture, there are three possibilities for communication between threads, marked *A*, *B* and *C* in the figure. Threads running on the same core, as in case *A*, can communicate through the fast L1 or L2 caches and have the highest communication

(a) Phase 1. (b) Phase 2.

Figure 1. The two phases of the producer-consumer benchmark. Circled threads communicate with each other.

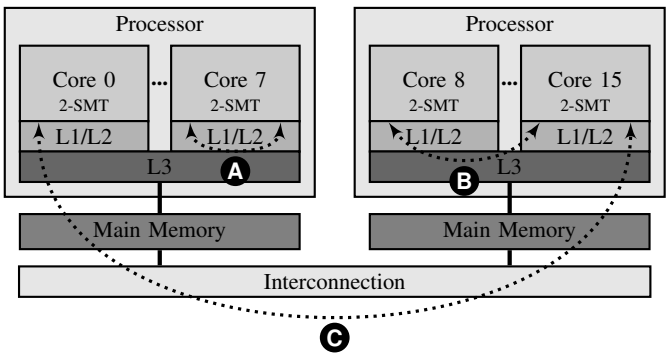
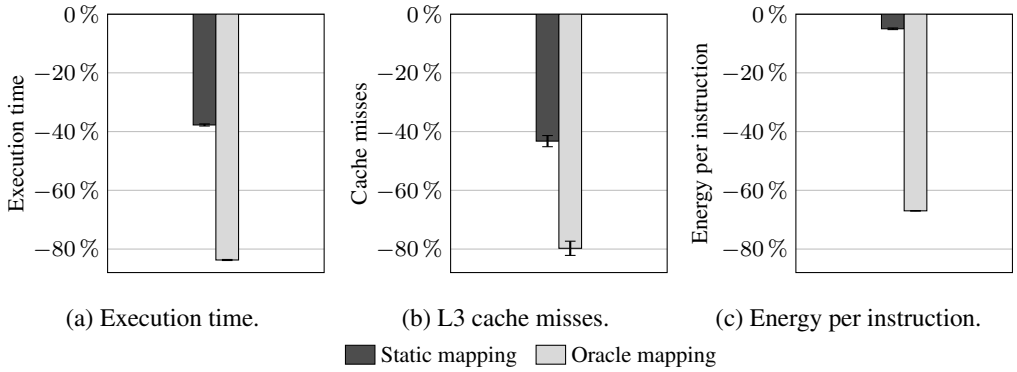


Figure 2. System architecture with two processors. Each processor consists of eight 2-way SMT cores and three cache levels. *A*, *B* and *C* show three different communication possibilities between two threads.

DOI: 10.1002/cpe



(a) Execution time. (b) L3 cache misses. (c) Energy per instruction.

Figure 3. Statistics of the producer-consumer benchmark, normalized to the operating system scheduler.

performance. In case *B*, threads that run on different cores have to communicate through the slower L3 cache, but can still benefit from the fast intrachip interconnection. When threads communicate across physical processors, as in case *C*, they need to use the slow off-chip interconnection. Hence, the communication performance in case *C* is the slowest of the three cases in this architecture.

The results using the producer-consumer benchmark are shown in Figure 3. All results are normalized to the values achieved by the operating system scheduler. We can observe that, in this communication bound application, communication-aware thread mapping provides a high reduction of both execution time and energy usage. The main reason for these improvements is the reduction of cache misses. The static mapping presents better results than the operating system because it correctly maps the threads for the first phase. This experiment demonstrates the potential gains by considering communication when mapping the threads of parallel applications. It also shows that dynamic mapping has more potential for improvements than static mapping, since it can react to changes in the communication pattern.

3. RELATED WORK

In this section, we contextualize the state of art in communication-aware thread mapping, covering both static and dynamic mechanisms. Several proposals present techniques to statically collect the communication pattern of the threads of parallel applications based on shared memory [5, 6]. Their methods consist of instrumenting simulators to generate memory accesses traces, which are analyzed to determine the communication pattern of the applications. The application is then executed with static mappings based in the detected patterns. Despite improving performance, these static thread mapping mechanisms have several disadvantages. They require a previous execution of the application to detect the communication pattern. Also, they are not able to correctly map applications that present different communication patterns along the execution. Furthermore, several mechanisms require simulation to detect the communication, and hence are infeasible for real applications.

Dynamic thread mapping mechanisms also have been proposed. Azimi et al. [10] show that hardware performance counters of current processors may be used to dynamically map parallel applications. They indirectly estimate the communication pattern using hardware counters of the Power5 processor. These hardware counters monitor memory accesses that are resolved by cache memories located in remote chips. Memory accesses resolved by local cache memories or the main memory are not considered when detecting the communication, generating an incomplete communication pattern.

Autopin [7] automatically selects the best mapping from a previously determined set of mappings, but requires the external generation of the mappings using other mechanisms. Autopin chooses the mapping with highest number of instructions per cycle (IPC). A similar mechanism exists

DOI: 10.1002/cpe

for applications that use software transactional memory [13], Radojkovic et al. [8] select the best mapping by measuring the performance that each mapping obtained. When the number of threads is low, all possible thread mappings are evaluated. When the number of threads makes it unfeasible to evaluate all possibilities, the authors execute the application with 1000 random mappings to select the best one. Hardware counters of current architectures can only be used to estimate the communication pattern between the threads indirectly and are very sensitive to interference from other applications.

The ForestGOMP library [9] integrates into the OpenMP runtime environment and gathers information about the different parallel sections of the applications. The threads of parallel applications are scheduled such that threads created by the same parallel region, which usually operate in the same dataset, execute on cores nearby in the memory hierarchy, decreasing the number of cache misses. The library depends on hints provided by the programmer, while our mechanism is performed automatically by the operating system. Su et al. [14] improve the performance of OpenMP applications by generating mappings considering hardware counters. As previously mentioned, hardware counters of current architectures are not accurate enough to estimate the communication pattern. Furthermore, these mechanisms only benefit OpenMP applications. On the other hand, our mechanism does not depend on any particular API and is more accurate because it has access to the memory pages that each thread is accessing.

Our previous work [11] introduced the basic concept of using the translation lookaside buffer to detect the communication pattern of a parallel application. The pattern is detected globally, during the whole execution of the application and is then used to calculate a static mapping of threads to cores for subsequent executions. SPCD [15] uses page faults of the parallel application to estimate the communication behavior, which can be detected in software. Due to the high software overhead, only few memory accesses can be considered for the communication pattern. Both mechanisms are limited to applications are limited to applications with a static communication behavior, as there is no provision for addressing changing behavior during execution. In [16], we present a mechanism to detect communication based on coherence message sent between cores. Although this mechanism supports dynamic changes in the communication behavior, it requires extensive hardware modifications, such as the addition of a small cache per core to count the number of invalidation messages. The total space required to store this cache has a quadratic complexity in the number of cores, and does not scale if the number of cores is large.

4. USING THE TRANSLATION LOOKASIDE BUFFER TO DETECT COMMUNICATION

Our communication detection mechanism uses the hardware implementation of virtual memory to perform the detection. Virtual memory requires the translation of virtual addresses to physical addresses for every memory access. To translate addresses, the operating system stores page tables in the main memory. However, to reduce the overhead imposed by the memory accesses to the page table, a special cache memory, called the *Translation Lookaside Buffer (TLB)*, is responsible for storing the page table translation entries for the most recently accessed pages. The TLB can be used to analyze the memory access behavior of parallel applications and thereby detect the communication between threads. This communication behavior can be used to map the threads of the application. In this section, we explain our proposal of using the TLB to detect the communication, starting with the concept of the mechanism, followed by the implementation.

4.1. Concept of the Detection Mechanism

In multi-core architectures, each core has its own TLB, which stores the most recently accessed page table entries by the core. If the same page table entry is present in more than one TLB, that means that the corresponding cores access shared memory at the page level granularity. If we iterate over all TLBs on the system and register every time two entries match, we get the amount of pages shared by the cores as a result. As the operating system knows which thread is executing in each core, this information can be used to detect the communication between the threads. By performing

DOI: 10.1002/cpe

Index	TLB Core 0	TLB Core 1	TLB Core 2	TLB Core 3
0x00	0x00	0x01	0x02	0x03
	0x48	0x41	0x41	0x49
0x01	0x10	0x11	0x12	0x13
	0x54	0x54	0x51	0x54
0x02	0x25	0x29	0x21	0x22
	0x66	0x60	0x63	0x67
0x03	0x37	0x37	0x32	0x30
	0x71	0x78	0x79	0x74

Figure 4. Detecting the communication between threads using the Translation Lookaside Buffer. Threads running on cores 1 and 2 communicate through page 0x41. Threads running on cores 0, 1 and 3 communicate through page 0x54. Threads running on cores 0 and 1 communicate through page 0x37.

this procedure systematically, we get a representation of the communication pattern at the page level granularity.

Figure 4 shows an example of how the mechanism finds communication, considering an architecture consisting of 4 cores. Each TLB contains 8 entries and is organized as a 2-way set associative cache. We can observe that some page entries are present in more than one TLB. Therefore, considering that a page present in more than one TLB represents a communication event, threads running on cores 1 and 2 communicate through page 0x41, threads running on cores 0, 1 and 3 communicate through page 0x54, and threads running on cores 0 and 1 communicate through page 0x37.

4.2. Implementation in Current Architectures

The implementation of the communication detection follows the steps outlined in Figure 5. To start our detection mechanism, the kernel can use several triggers, which will be discussed in Section 4.3.

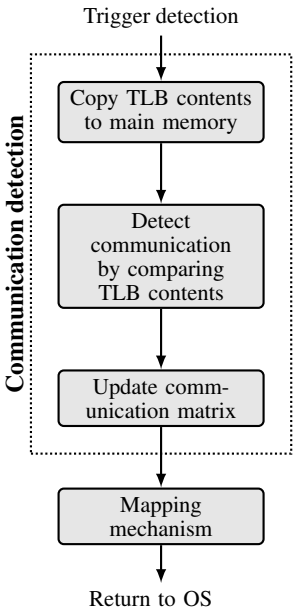


Figure 5. Overview of the proposed mechanism. The communication detection is described in Sections 4.1 and 4.2, the trigger in Section 4.3. The communication matrix is then fed to the mapping mechanism, explained in Section 5.

DOI: 10.1002/cpe

When the mechanism is triggered, we first access the contents of the TLBs. Accessing remote TLBs from a specific core could harm the performance. To avoid that, the kernel has to access only the local TLB. By periodically doing this from all cores, the kernel can keep a mirror of the TLBs in the main memory. This mirror is then used to detect the communication between the threads by comparing the TLB contents. The final step is to store the communication for further processing. In this section, we first introduce the data structure used to store the amount of communication between the threads. Then, we describe the implementation of the mechanism in current architectures.

4.2.1. Communication Matrix Communication can be analyzed by grouping different numbers of threads. To calculate the amount of communication between groups of threads of any size, the time and space complexity rises exponentially with the number of threads. To keep the complexity quadratic, we evaluated the communication between pairs of threads, generating a *communication matrix*, where each cell (i, j) contains the amount of communication between threads i and j. In our mechanism, the amount of communication is related to the amount of pages shared by the threads. Grouping larger numbers of threads is supported by the detection mechanism without any changes. We do not consider the diagonal of the matrix, since it contains accesses to memory regions that are private to a thread, and therefore do not represent communication between threads.

4.2.2. Implementation in Hardware-managed TLBs In most current processor architectures, such as Intel x86/x86-64 [17] and ARM [18], the contents of the TLB are managed by the hardware. For every memory access, the TLB is searched for a match. If the corresponding entry is cached in the TLB, the address is translated and sent to the memory hierarchy. If the entry is not present in the TLB, the hardware accesses the page table in main memory and loads the corresponding entry into the TLB. This mechanism is called *hardware-managed* TLB. The only operation that is performed by the operating system in this type of TLB is the management of the page table and the invalidation of TLB entries when the page table is modified.

To allow finding the communication pattern, architectures with hardware-managed TLBs require a minor change to the hardware, as the operating system does not have access to the contents of the TLB. The modification consists of adding an instruction that allows the operating system to access the contents of the TLB. Using this instruction, the kernel can search the TLBs for matching entries periodically. Our detection mechanism compares the contents of all TLBs, incrementing the communication matrix for each match. When the same address is found in TLBs i and j, the communication matrix is incremented in cells (i, j) and (j, i).

4.2.3. Implementation in Software-managed TLBs In some RISC architectures, such as SPARC [19] and MIPS [20], the processor traps to the operating system when a TLB miss occurs. The operating system then accesses the page table in the main memory and loads the corresponding entry into the TLB. This type of TLB is called a *software-managed* TLB. To implement a mechanism to detect the communication pattern for the software-managed TLB, no hardware modification is required. When a TLB miss traps to the operating system, the kernel can also check the contents of the other TLBs for matches besides loading the entry from the main memory, incrementing the communication matrix whenever a matching entry is found.

4.2.4. Comparing the TLB Contents For the communication detection, we need to compare all pairs of TLBs. Two different comparison strategies can be adopted. To simplify the explanation, we consider that the number of threads is equal to the number of cores. The first strategy is to use only one thread to compare all pairs of TLBs. If the TLB is fully associative, the time complexity for the algorithm to find the communication is $\Theta(T^2 \cdot E^2)$, where T is the number of threads and E is the number of entries in each TLB. The complexity is quadratic in T because it is necessary to compare every possible pair of TLBs. The comparison of all the entries of the pair of TLBs is quadratic in E. However, if we use a set associative TLB, the time complexity is decreased to $\Theta(T^2 \cdot E)$ as it is not necessary to check all entries of the TLB for matches, but just the entries that are on the same set. We consider that the associativity is a constant, because it is much smaller than the number of

DOI: 10.1002/cpe

entries in the TLB. The second strategy consists of each thread comparing its own TLB to the other TLBs. In this way, the time complexity of the detection mechanism decreases to $\Theta(T \cdot E)$ for a set associative TLB, since each thread computes in parallel. Therefore, in this paper, we use the second strategy.

4.2.5. Multi-threaded Architectures In multi-threaded architectures, such as Simultaneous Multi-Threading (SMT), the TLB may be shared between the virtual cores. If the multi-threaded architecture features one TLB per virtual core, our mechanism works as previously described. If the TLB is shared between the virtual cores, it contains bits to identify which of the virtual cores accessed each entry. In this case, our detection mechanism needs to compare these bits as well to detect an accurate pattern. Since this is a very short operation, it does not increase the overhead.

4.2.6. Multi-level TLBs Some architectures contain several levels of TLBs, working in the same way as the cache memory hierarchy. To support multi-level TLBs, our mechanism just needs to read the contents of the TLB from the desired level. Usually, the higher levels have less entries and represent the most recent accessed data. By using the contents of the higher level TLB, there will be less entries considered for matching. On the other hand, by using the contents of the last level TLB, there will be more entries to be considered for matching. Several levels may be considered if the TLB hierarchy is not inclusive.

4.3. Triggering the Detection Mechanism

Several methods can be used to trigger the detection mechanism. We introduce three triggers and evaluate their benefits and drawbacks. We also calculate the time complexity of each trigger, combined with the detection strategy, considering a set associative TLB. In the calculations of the complexities, consider that T is the number of threads and E is the number of entries of each TLB. To simplify the explanation, we consider that the number of threads is equal to the number of cores. The first three triggers can be used with both software-managed and hardware-managed TLBs, the last one only with software-managed TLBs.

For the first three triggers, we need to compare all pairs of TLBs, which has a time complexity of $\Theta(T \cdot E)$ for a set associative TLB. For the fourth trigger, which only works in architectures with software-managed TLBs, the time complexity to find the communication in a fully associative TLB is $\Theta(T \cdot E)$. The complexity increases linearly with T since we have to check all the other TLBs. It is also linear with E because all the entries of the TLB are searched for matches. However, considering a set associative TLB, the time complexity of the detection mechanism for the forth trigger decreases to $\Theta(T)$.

4.3.1. Trigger based on Time Intervals It is possible to trigger the detection mechanism at fixed time intervals. However, using the time as trigger is not a good solution, because it does not gather any knowledge of the application. For instance, consider the case that the sampling is made when the threads 0 and 1 are accessing their shared data, but at the same time the other threads are working on their private data. As this situation describes a temporary behavior, it may not characterize the global behavior of the application. If this happens several times, the mechanism will detect a lot of communication between threads 0 and 1, but none for the other threads. This does not imply that the other threads do not communicate, it means that it was not possible to find the global communication at the time of sampling.

The amount of times the mechanism is triggered is given by:

$$\frac{ExecTime}{TriggerPeriod} \tag{1}$$

Therefore, the time complexity for the detection mechanism using time as trigger is:

$$\Theta\left(\frac{ExecTime}{TriggerPeriod} \cdot T \cdot E\right) \tag{2}$$

DOI: 10.1002/cpe

4.3.2. Trigger based on Global TLB Misses We can analyze the memory usage of an application by monitoring the number of TLB misses generated by it. This is possible in most architectures, since a hardware counter that counts TLB misses is usually present. The occurrence of a TLB miss indicates that the application is changing the ranges of the virtual address space being used. Therefore, if the number of TLB misses is greater than $T \cdot E$, it is likely that the application changed all TLB entries. Triggering the detection mechanism at this point provides a better result than using the time based trigger, as the contents of the TLBs should have changed enough to represent a new communication pattern. The time complexity of the detection mechanism using this trigger is:

$$\Theta\left(\frac{TotalTLBmiss}{T \cdot E}\right) \cdot \Theta\left(T \cdot E\right) \tag{3}$$

As the global number of TLB misses is equal to the average misses per TLB, multiplied by the number of TLBs, we get:

$$\Theta\left(\frac{TLBmissPerThread \cdot T}{T \cdot E}\right) \cdot \Theta\left(T \cdot E\right) \tag{4}$$

Simplifying this equation leads to a complexity for the detection mechanism using the global TLB misses trigger of:

$$\Theta(TLBmissPerThread \cdot T) \tag{5}$$

4.3.3. Trigger based on Local TLB Misses The global TLB misses based trigger provides a good accuracy, but its time complexity depends on the number of threads. This is not desired, as it is expected that the number of cores and threads per system will increase in the future. To overcome this issue, instead of keeping a global counter of TLB misses, each thread keeps track of only its own TLB misses. In this way, the time complexity depends only in the number of TLB misses. The complexity of the detection mechanism using this trigger is:

$$\Theta\left(\frac{TLBmissPerThread}{T \cdot E}\right) \cdot \Theta(T \cdot E) \tag{6}$$

Simplifying this equation leads to a complexity for the detection mechanism using the local TLB misses trigger of:

$$\Theta(TLBmissPerThread) \tag{7}$$

4.3.4. Trigger based on Shared Pages In software-managed TLBs, it is possible to keep track of which pages are shared between threads. Therefore, we can trigger the detection mechanism only for TLB misses of shared pages, since pages private to one thread are not used for communication. To identify shared pages, we can store the sharer threads of each page on the page table. Therefore, whenever a thread accesses a page, it becomes a sharer of that page. When the page table entry is evicted from the TLB of the core that is executing the thread, the thread is removed from the corresponding sharers list. All pages that have more than one sharer thread are considered shared, and any TLB miss on that page triggers the detection mechanism. To calculate the complexity of this trigger, we consider that the ratio of shared pages relative to the total amount of pages is a constant factor. The complexity of the detection mechanism using this trigger is:

$$\Theta(TLBmissPerThread \cdot T) \tag{8}$$

4.3.5. Summary of the Triggers From the triggers described, we used the trigger based on local TLB misses. This trigger can be used in both software-managed and hardware-managed TLBs, and is able to adapt to each application because it has knowledge about when an application may be changing its memory access pattern through the TLB misses. Also, it presents a low overhead because it is independent from the number of cores and threads.

DOI: 10.1002/cpe

4.4. Design Considerations

As our mechanism is performed entirely by the hardware and the operating system, it does not depend on the parallelization API and does not require any modification to the application. Moreover, the communication pattern is detected only during the execution of the application. Our mechanism provides a good solution for detecting changes in the behavior of the applications because the number of possible entries in the TLB is quite low. Data that is not accessed anymore will have its corresponding entry overwritten and will therefore not be counted anymore in the calculation of the communication pattern.

Some architectural parameters influence the behavior of our mechanism. The higher the number of entries of the TLB or its associativity, the higher the number of entries considered when looking for matches. This increases the accuracy, but also the overhead. However, the TLB miss rate also depends on these parameters, such that the higher the number of entries or associativity of the TLB, the lower the TLB miss rate. To avoid losing accuracy when the TLB miss rate is low, the only thing needed is to adjust the trigger to enable the detection with less TLB misses. Any access to the same page is considered as communication, therefore increasing the page size could lead to higher false sharing if the memory usage is constant. However, the memory usage of parallel applications is also increasing proportionally, which mitigates this problem. A larger page size also reduces the TLB miss rate, but this is easily overcome by adjusting the trigger, as previously described.

5. THE MAPPING MECHANISM

The communication detection presented in the previous section generates a communication matrix. This matrix is used by the mapping algorithm to calculate an optimized assignment of threads to cores. The mapping problem is NP-Hard [21]. Consequently, finding the optimal solution becomes infeasible when the number of threads grows. Heuristic algorithms can be employed to determine the mapping in reasonable time, with results close to the optimal mapping.

Related work describes several heuristic algorithms [22, 23]. These algorithms focus on thread mapping using general graph algorithms, while mapping can be performed with a more specific tree-based structure, as the memory hierarchy can be represented by a tree. A tree-based approach

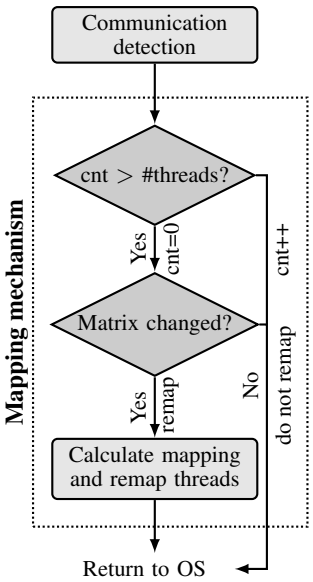


Figure 6. Overview of the mapping mechanism. The communication matrix is detected as explained in Section 4, and is fed to the mapping mechanism. The communication filter is responsible for evaluating if the communication matrix changed (third box from the top).

DOI: 10.1002/cpe

was proposed in [24]. For each level on the memory hierarchy, it presents a time complexity of $O(T^N)$, where T is the number of threads and N is the number of sharers of the corresponding level, because it generates all possible groups of threads for a given level of the hierarchy. This leads to a high overhead for architectures with a large number of sharers.

We developed a thread mapping algorithm based on the graph matching problem. Our solution uses the graph matching algorithm from Edmonds [25], which has a polynomial time complexity. The mapping algorithm is triggered by the same mechanism as the communication detection, as shown in Figure 6. To reduce the overhead, we use a counter (cnt) that calls the next step only if the trigger was activated more times than the number of threads. To further reduce the overhead of the mapping algorithm, we developed an algorithm that aims to decrease the amount of times that the mapping algorithm is called. We refer to this algorithm as *communication filter*. In this section, we first describe this communication filter and then the mapping algorithm itself.

5.1. Communication Filter

The goal of this filter is to decide if the communication matrix has changed sufficiently to warrant a migration of threads. The filter is based on the idea that each thread communicates more with a certain subgroup of threads. We call the threads that belong to the same subgroup *partner threads*. In the communication filter, we limit the size of each subgroup to 2. In this way, the overhead is reduced, since we only keep track of one partner thread for each thread.

Every time a new communication matrix is evaluated, the algorithm counts how many threads changed their partner. If the amount of different partners exceeds a given threshold, the mapping algorithm is called. Otherwise, the communication filter algorithm considers that the communication matrix did not change enough to represent a new communication pattern. We used 2 as threshold, since if 2 threads changed partners, it probably means that they started to communicate with each other. The time complexity of this algorithm is $\Theta(T^2)$, where T is the number of threads of the application.

5.2. The Thread Mapping Algorithm

Our algorithm to map the threads on the cores is based on maximum weight perfect matching problem for complete weighted graphs. This problem is defined as follows. Given a complete weighted graph $G = (V, E)$, we have to find a subset M of E in which every vertex of V is incident with exactly one edge of M, and the sum of the weights of the edges of M is maximized. The Edmonds' matching algorithm solves this problem in polynomial time [25], $O(N^3)$, and a parallel algorithm can solve the problem with a time complexity of $O(\frac{N^3}{P} + N^2 \lg N)$, where N is the number of vertices and P is the number of processing elements.

We model thread mapping as a maximum weight perfect matching problem as follows. Vertices represent threads. Edges represent the amount of communication between threads. A complete graph is obtained directly from the communication matrix, as in Figure 7a. The graph is processed by the matching algorithm, which outputs the pairs of threads such that the amount of communication is maximized, illustrated in Figure 7b.

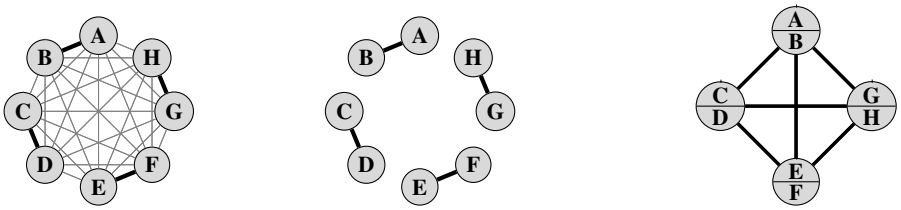
To support multi level hierarchies and architectures in which there are more than 2 cores sharing a cache, the matching algorithm must be executed several times. On every execution, another communication matrix needs to be generated, in which each vertex represents previously grouped threads, and the edges represent the communication between the corresponding groups, depicted in Figure 7c. This matrix was generated by function H:

$$H_{(x,y),(z,k)} = M_{(x,z)} + M_{(x,k)} + M_{(y,z)} + M_{(y,k)} \tag{9}$$

where (x, y) and (z, k) are the matches found at the previous step, and $M_{(i,j)}$ is the amount of communication between threads i and j found in the communication matrix M. Then, the matching algorithm is re-executed using this new communication matrix as input.

This procedure is repeated $\log_2(N)$ times in each level of the memory hierarchy, where N is the amount of sharers of the corresponding level. Therefore, the complexity of our mapping algorithm

DOI: 10.1002/cpe



(a) Input. (b) Output. (c) Generate new matrix.

Figure 7. The graph matching problem applied to thread mapping. Each vertex corresponds to a thread or a group of threads. Edges represent the amount of communication between them.

(a) Group only the threads that maximize communication. (b) Generate a bipartite graph.

Figure 8. Mapping algorithm when the number of cores sharing a cache is 3. Each vertex represents a thread or a group of threads. Edges represent the amount of communication between them.

is $O(T^3 \log_2(N))$ for each level, where T is the number of threads. For instance, if there are 4 cores sharing the L1 cache, this procedure is repeated 2 times to calculate which threads will share the L1 cache. The same procedure is applied for each level on the memory hierarchy. For instance, if the L1 cache is shared by 2 cores, and the L2 cache is shared by 2 L1 caches, the graph matching is applied twice. The first matching generates the pairs of threads that will share the L1 cache. The second matching generates the pairs of pair of threads that will share the L2 cache.

The number of cores sharing a cache may not be a power of two. In this scenario, we also apply the matching more than once per cache level. Considering an architecture with 6 cores and 2 caches, where each cache is shared by 3 cores, the matching algorithm would produce 3 disconnected graphs that represent 3 pairs of threads. We generate a new communication graph that groups only the 2 pairs that maximize the communication, as in Figure 8a. Afterwards, we insert edges connecting the vertices in such a way to form a bipartite graph that puts the grouped threads in the same set, as in Figure 8b. Finally, we apply the matching to the bipartite graph. This procedure can be adapted to map any number of threads.

6. EXPERIMENTAL EVALUATION

We begin this section by explaining our methodology. Afterwards we show how we validated our mechanism. Then, we show the the detected communication patterns, as well as the performance and energy results. Finally, we evaluate the overhead of our mechanism and compare it to related work.

6.1. Methodology

Our experiments were performed on a real and on a simulated machine. We describe these two evaluation environments, as well as the benchmarks used. In both environments, we use the *Local TLB misses* trigger, described in Section 4.3, because it presents the best trade-off between accuracy and overhead for architectures with hardware-managed TLBs. Table I summarizes the parameters of the real and simulated machines.

DOI: 10.1002/cpe

Table I. Real and simulated machine configuration.

	Parameter	Value
Real machine	Processors	2x Xeon E5-2650 (Sandy Bridge-EP), 2.0 GHz, 8 cores, 2-SMT
	Caches / processor	8x 32 Kbytes L1, 8x 256 Kbytes L2, 20 Mbytes L3
	Memory	32 Gbytes DDR3-1333, 4 Kbytes page size
	TLBs	64/512 entries (L1/L2)
Simics	Processor	2.0 GHz, 32 nm, 4 Kbytes page size
	L1 cache	32 Kbytes, 4-way, 2 cycles latency
	L2 cache	8 Mbytes, 16-way, 6 cycles latency
	Cache coherence	snoop-based MESI protocol, 64 bytes per line
	TLBs	64 entries, 20 cycles to read entry

6.1.1. Experiments in a Full System Simulator We used the Simics full system simulator [26] to evaluate our proposal. The simulated machine has 4 sockets, each with 2 cores, with private L1 caches, and L2 caches shared by all cores. Cache latencies were calculated using CACTI [27]. To fetch the contents of the TLB, we added an instruction by using the *magic instruction* extensions of Simics. In this experiment, we used only one TLB level, with 64 TLB entries per TLB.

6.1.2. Experiments in a Real Machine The real machine consists of 2 Intel Xeon E5-2650 processors, with a total of 32 SMT cores. The memory hierarchy of the system is illustrated in Figure 2. Since this system uses hardware-managed TLBs and does not implement an instruction to allow the kernel to access the contents of the TLB, we generate a simulation trace with information about the communication in Simics. The simulation trace is then fed into the mapping mechanism during the execution of the applications on the real machine to perform the thread mapping. In this experiment, we did not configure a memory hierarchy in Simics to allow us to simulate benchmarks with higher input sizes. Also, we implemented the detection mechanism in the second level TLB, which has 512 entries shared among the virtual cores. The mapping code was implemented as a wrapper for the OpenMP and Pthreads libraries. We gather information about the memory hierarchy of the real machine using the Hwloc library [28] during the initialization of our mechanism.

6.1.3. Benchmarks As workload, we used the OpenMP implementation of the NAS Parallel Benchmarks [29], version 3.3.1, and the PARSEC Benchmark Suite [30], version 3.0. The applications of NAS were executed using the *W* input size for the experiments in the simulator, and with the *A* input size for the experiments in the real machine. PARSEC was executed only in the simulator, with the *simlarge* input size. We did not execute DC, from NAS, and PARSEC in the real machine to the high usage of dynamic memory, which invalidates the simulation trace used to guide mapping in the real machine. Larger inputs were not executed due to simulation time constraints. We run the benchmarks with one thread per virtual core. Applications of PARSEC create more threads than the number of virtual cores, such that more than one thread can be mapped to the same virtual core.

6.2. Validating the Mechanism with the Producer-Consumer Benchmark

We validated our proposal with the producer-consumer benchmark introduced in Section 2 with 16 pairs of producer-consumer threads. The resulting communication matrices are shown in Figure 9. In this figure, the thread IDs of the threads are shown at the x and y axis. Communication between each pair of threads is shown in the cells, where a dark color indicates more communication.

Our mechanism is able to identify the two distinct communication phases of the application. The first phase is shown in Figure 9a, where the communication is performed between neighboring threads. The second phase is depicted in Figure 9b, where the communication is performed between

DOI: 10.1002/cpe

threads with the same congruence modulo 16. During the transition between the two phases, the detected communication is illustrated in Figure 9c, which contains traces of both phases.

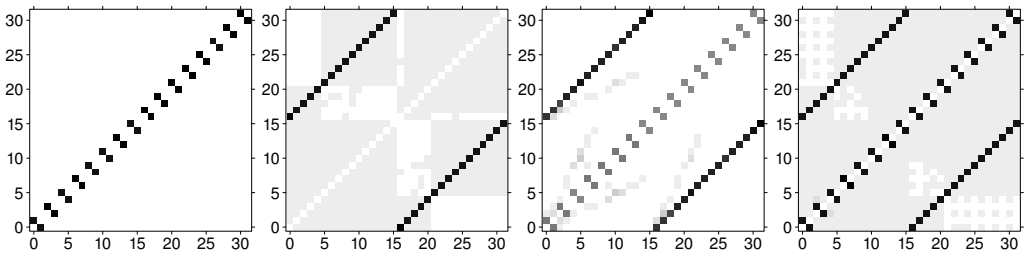
These results verify that our proposed mechanism is able to detect communication patterns correctly during the execution of a parallel application, and can react to changes in the communication behavior. If a static detection mechanism was used, it would not be able to handle the different behaviors of the two phases, since it would have access only to the overall communication pattern as shown in Figure 9d.

6.3. Communication Patterns of NAS and PARSEC

The communication matrices of the NAS-OMP and PARSEC benchmarks obtained with our mechanism are shown in Figure 10. 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 a larger amount of communication between threads. NAS applications were executed with 32 threads in the real machine and 8 in the simulator. PARSEC applications were configured to execute with 8 threads in the simulator, but the actual amount of threads created vary from 8 to 256 depending on the application.

We can divide the applications into two main groups: *heterogeneous* and *homogeneous* communication patterns. In applications that presents heterogeneous communication patterns, there are threads that communicate more with a subgroup of threads. BT, IS, LU, SP and UA present heterogeneous communication patterns. Most communication happens between directly neighboring threads, which is common when the application is based on domain decomposition, where most of the shared data is located on the borders of each sub-domain. These patterns show dark cells near the diagonal of the communication matrices, and nearly white cells farther away from the diagonal. Swaptions has a similar pattern, but with less expressive communication. MG also presents communication between neighbors, but the degree of heterogeneity is lower and the communication between more distant neighbors is more evident than in the other applications. Fluidanimate and x264 also have communication between more distant threads, although it is not very clear in Figure 10n due to the high number of threads (256 threads). x264, Ferret and Dedup have a pipeline communication behavior. We can identify in Ferret and Dedup that threads that communicate in the pipeline form communication clusters.

When each pair of threads in an application has a similar amount of communication, we classify this application as having a homogeneous communication pattern. CG, EP, FT and Canneal present homogeneous communication patterns. In CG and FT, although there are small differences in the detected amount of communication, there are no distinct subgroups of threads that maximize the communication if mapped nearby in the memory hierarchy. The communication pattern of Vips is similar to the one of CG. In EP, besides being homogeneous, there is almost no communication between the threads. In homogeneous applications, any thread mapping would result in the same performance. Therefore, we expect benchmarks that have a heterogeneous communication pattern to have higher improvements when using communication-aware mapping.



(a) Phase 1. (b) Phase 2. (c) Phase transition. (d) Overall behavior.

Figure 9. Communication pattern of the producer-consumer benchmark. Darker areas indicate more communication between threads.

DOI: 10.1002/cpe

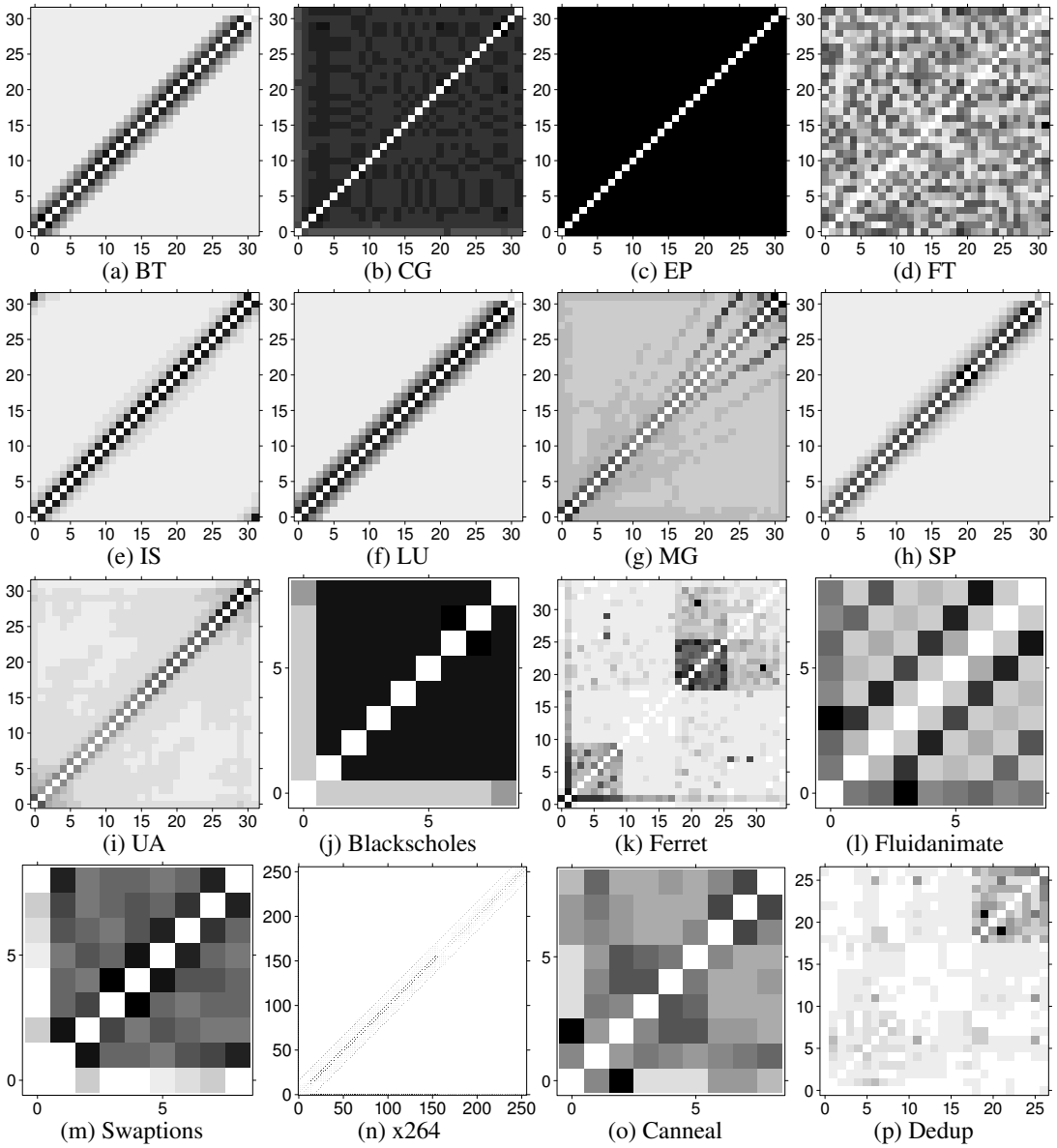


Figure 10. Overall communication patterns detected by the TLB mechanism. Darker cells indicate more communication between threads.

6.4. Performance Experiments

We performed experiments in the simulator and real machine. In the simulator, we analyzed execution time. In the real machine, we analyzed execution time, L2/L3 cache MPKI and cache-to-cache transactions. L2/L3 MPKI were measured with Intel PCM [31]. The number of cache-to-cache transactions was measured with Intel VTune [32], using the `OFFCORE_RESPONSE.ALL_DEMAND_MLC_PREF_READS.LLC_MISS.REMOTE_HITM_HIT_FORWARD` event.

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. In the simulator, we compare to the operating system and to a round-robin mapping. The *operating system mapping* uses the unmodified scheduler of the Linux kernel, version 3.5. It represents the baseline for our

DOI: 10.1002/cpe

Table II. Absolute values of the results of the experiments achieved by the TLB mechanism. The difference to the operating system mapping is shown in parentheses.

	Parameter	BT	CG	EP	FT	IS	LU	MG	SP	UA
Performance	Execution Time (s)	5.29 (-9.2%)	0.17 (+2.2%)	0.95 (-0.3%)	0.77 (+0.3%)	0.18 (-0.3%)	4.40 (-7.4%)	1.06 (-0.4%)	5.18 (-16.7%)	4.77 (-4.6%)
	L2 cache MPKI	0.82 (-23.1%)	5.37 (-2.3%)	0.08 (-2.5%)	2.77 (-0.1%)	2.60 (-3.2%)	1.07 (-23.3%)	3.65 (-0.8%)	2.11 (-25.1%)	0.91 (-24.4%)
	L3 cache MPKI	0.17 (-54.4%)	0.31 (-3.2%)	0.02 (-4.6%)	1.10 (-0.0%)	2.49 (-3.2%)	0.44 (-35.4%)	2.80 (-1.1%)	0.47 (-59.3%)	0.28 (-54.7%)
	Cache-to-cache transactions (k)	52706 (-67.0%)	1127 (-23.1%)	43 (+8.8%)	499 (-8.6%)	1 (-16.5%)	174322 (-49.3%)	1085 (-65.7%)	67465 (-76.3%)	133836 (-60.3%)
Energy	Processor energy (J)	640.86 (-6.8%)	19.52 (+1.9%)	99.98 (-0.3%)	87.63 (+0.5%)	18.55 (+0.1%)	494.77 (-6.9%)	106.79 (-0.4%)	583.35 (-14.8%)	517.06 (-4.6%)
	DRAM energy (J)	33.65 (-17.4%)	1.01 (+1.0%)	4.98 (+0.6%)	8.83 (-0.2%)	1.77 (-0.6%)	35.39 (-17.6%)	13.43 (0.0%)	49.99 (-24.3%)	28.87 (-23.6%)
	Energy per inst. (nJ)	1.80 (-6.5%)	2.31 (+1.7%)	2.16 (-0.4%)	3.52 (+0.2%)	3.98 (-1.0%)	2.34 (-7.0%)	8.45 (-0.2%)	2.89 (-13.2%)	2.36 (-5.9%)
	Number of migrations	1	1	1	3	1	1	2	1	1

experiments. For the *random mapping*, we generated 50 different mappings randomly, one for each execution, and execute all benchmarks with the same random mappings. This mapping presents an evaluation without the overhead of migrations introduced by the operating system. For the *oracle mapping*, we generated traces of all memory accesses for each application and perform an analysis of the communication pattern, similarly to [6]. This mapping optimizes the execution of each application in terms of the communication between threads. For the *round-robin* mapping, we map each thread to a core of the memory hierarchy following a compact strategy.

Figure 11 shows the execution time in Simics. Figure 12 shows the average results for execution time, L2/L3 cache MPKI and the number of cache-to-cache transactions in the real machine. Each experiment in the real machine was performed 50 times. We also show a 95% confidence interval calculated with the normal distribution The results are normalized to the values of the original scheduler of the operating system. Table II shows the absolute values for our mechanism in the real machine.

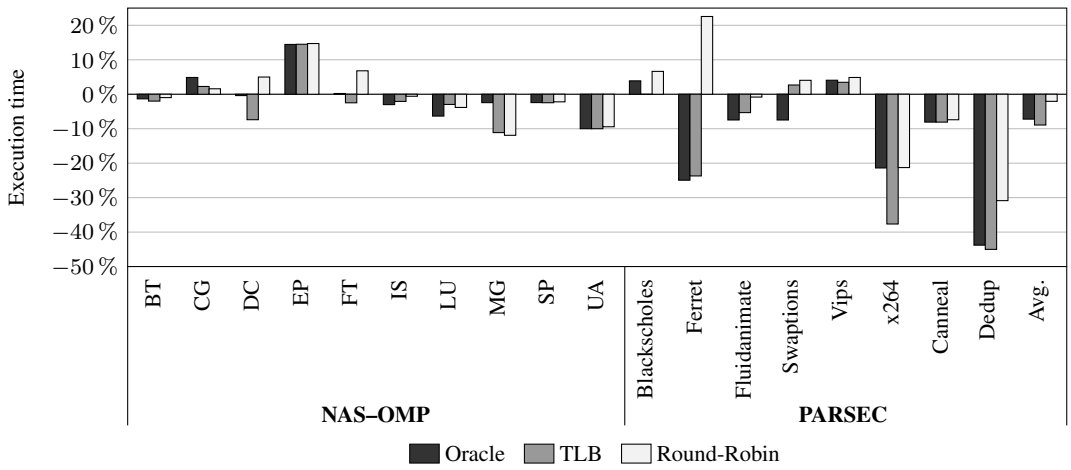


Figure 11. Performance results in Simics. All values are normalized to the results of the operating system mapping.

DOI: 10.1002/cpe

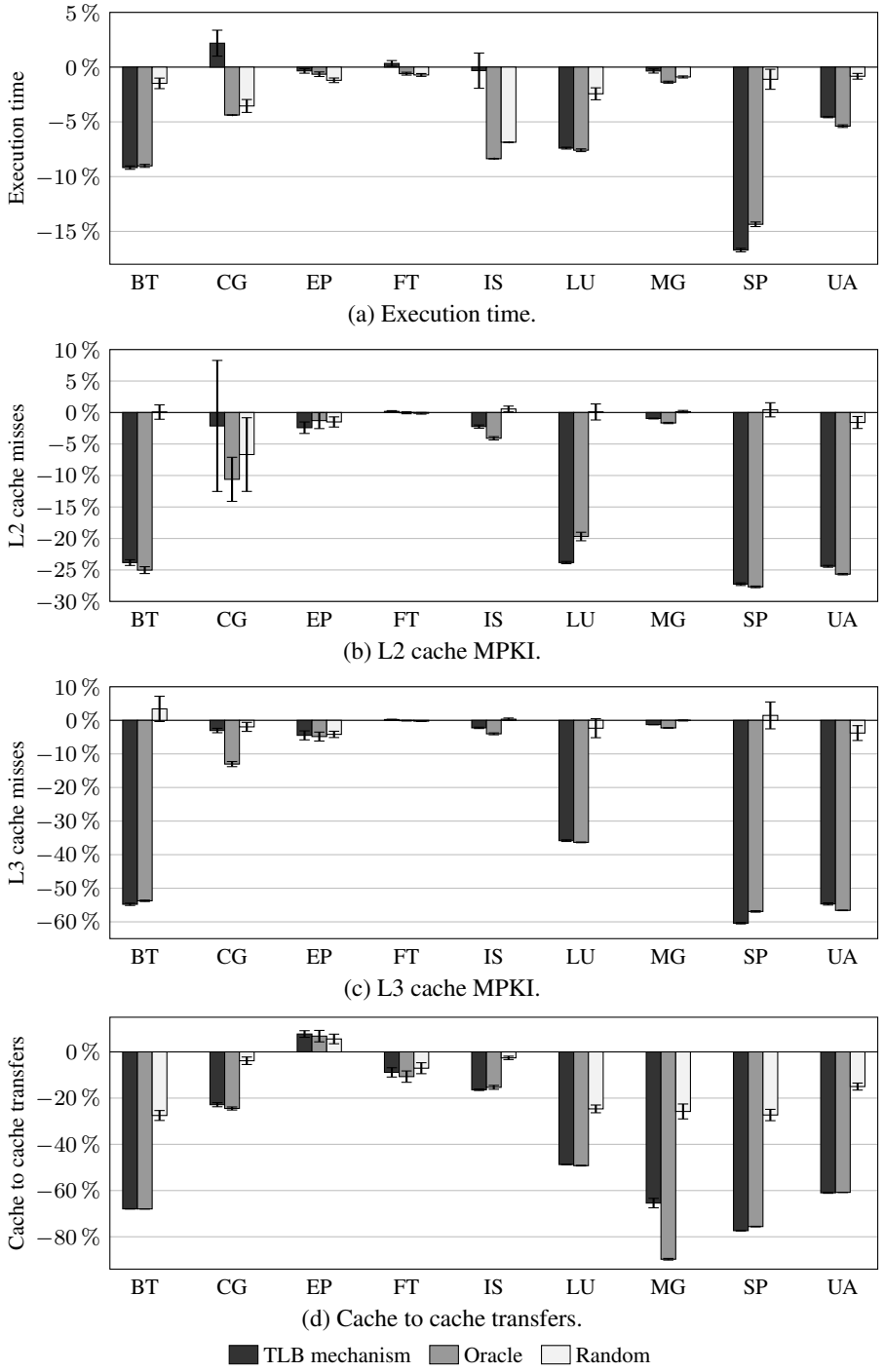


Figure 12. Performance results on the real machine. All values are normalized to the results of the operating system mapping.

The performance improvement was higher in applications with high heterogeneous communication patterns. In the real machine, BT and SP achieved the best results, reducing execution time by up to 9.2% and 16.7%, respectively. The performance improvements of these applications in the simulator was lower due to the lower number of threads, 32 against 8. Furthermore, the real machine has more levels in the memory hierarchy to be exploited: shared

DOI: 10.1002/cpe

L1 and L2 caches between SMT cores, shared L3 cache between all cores within a processor, as well as the intra-chip and inter-chip interconnections. As expected, the performance difference between a bad and a good mapping is more evident in complex memory hierarchies. In the simulator, Ferret, x264 and Dedup achieved the highest improvements, 23.7%, 37.6% and 45%. As previously described, these applications communicate using a pipeline model, and create many more threads than the number of cores. In this way, the complexity is increased, such that we were able to achieve good performance improvements in the simulator.

The number of cache-to-cache transactions and cache misses in the highly heterogeneous applications (BT, LU, SP and UA) were reduced significantly. The execution time also was reduced, but by a smaller amount. Also, our TLB mechanism achieved results similar to the oracle mapping, demonstrating its effectiveness. It is important to note that the number of cache-to-cache transactions is much more sensitive to thread mapping than cache misses. The reason is that a better mapping directly influences the number of cache-to-cache transactions, while cache misses are also influenced by other factors, such as cache line prefetching and competition for cache lines by the cores that share the cache, among others. This is the reason that cache misses have larger confidence intervals than cache-to-cache transactions. Likewise, the number of cache-to-cache transactions is also more sensitive to thread mapping than the execution time.

If the communication pattern among the threads is homogeneous, as in CG, EP and FT, no performance improvement is expected when mapping the threads according to the communication. EP, besides having a homogeneous communication pattern, shares almost no data between the threads, which is the reason that the absolute values for cache misses and cache-to-cache transactions are low compared to the other applications. These low values imply that small, unpredictable changes in the runtime behavior can have a big impact on the statistics, resulting in large confidence intervals. The improvements achieved in these homogeneous applications happened due to a reduction of the number of migrations, since the operating system scheduler unnecessarily migrates the threads several times during execution. In MG, only the L2 misses and cache-to-cache transfers were improved, since the communication pattern is less heterogeneous.

Summarizing the results, we can see that improvements were dependent on the way the applications used the shared memory. As expected, applications that communicated more and present heterogeneous communication patterns showed the largest improvements. Applications with homogeneous communication patterns presented only small improvements. This is the expected result, as there is no difference in the communication among the threads to be exploited. Our mechanism presented results similar to the Oracle mapping. This indicates that our mapping algorithms were effectively using the detected communication pattern. In most cases, our mechanism performed better than the random mapping. This shows that our 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 resources of the machine.

6.5. Energy Experiments

We also evaluated how our mechanism affects the energy consumption. As in the performance experiments, we execute each test 50 times, use a 95% confidence interval calculated with normal distribution, and compare the results of our proposal to the default mapping performed by the operating system, to random static mappings and to an oracle mapping. We measure both package and DRAM energy consumption. Besides the total amount of energy consumed, we also measured the amount of energy per instruction. To monitor the energy consumption, we used Intel PCM [31] to access the Running Average Power Limit (RAPL) hardware counters, introduced by Intel in the SandyBridge architecture [17].

Total processor and DRAM energy consumption is shown in Figures 13a and 13b. We can observe that the behavior is similar to the execution time results, with reductions for BT, LU, SP and UA. The other applications show no difference or a small reduction of energy efficiency. We can also observe that the DRAM energy was reduced more than the processor package energy. The results of energy per instruction in Figure 13c show that our mechanism not only saves energy by reducing

DOI: 10.1002/cpe

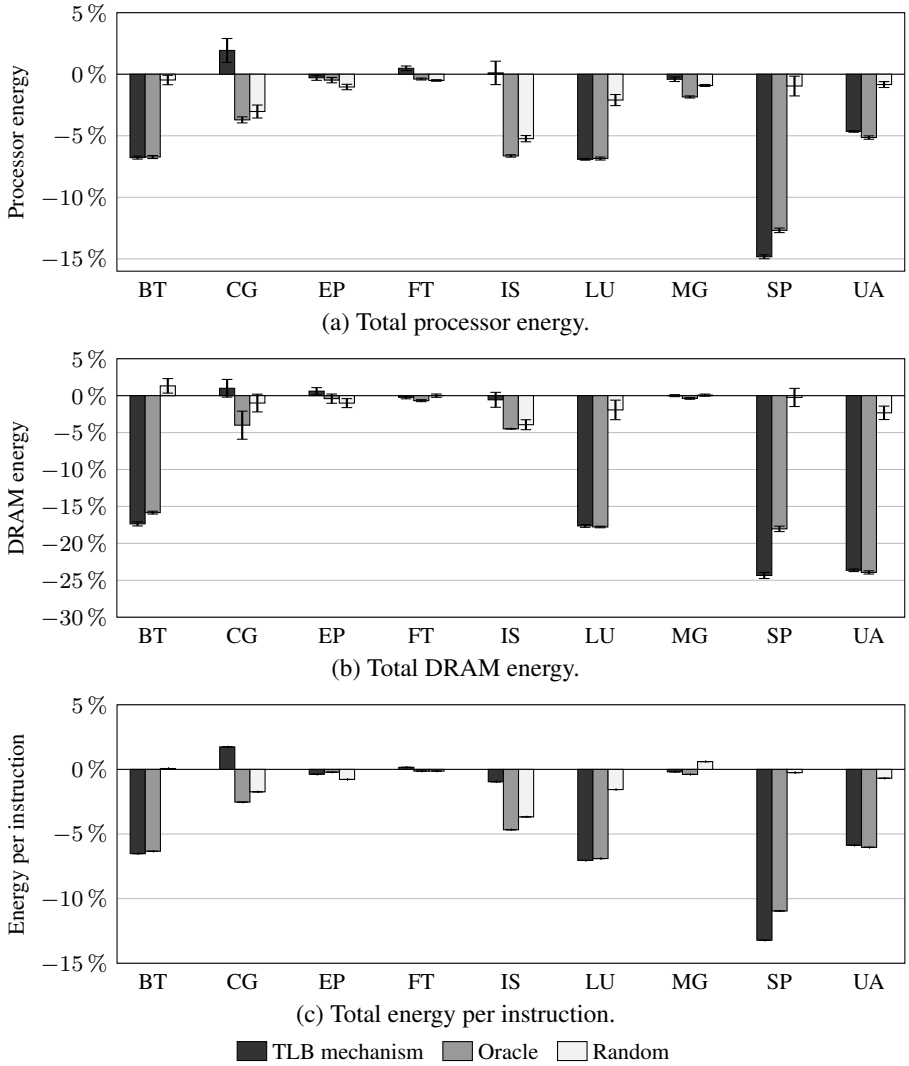


Figure 13. Energy consumption results. All values are normalized to the results of the operating system mapping.

the executing time, but also by providing a more efficient execution, which is an important goal for future Exascale architectures [33].

As in the performance experiments, applications with heterogeneous communication patterns present a higher reduction of energy consumption than applications with homogeneous communication patterns.

6.6. *Overhead of the Mechanism*

The overhead of our proposal is due to three main reasons: copying TLB entries to the main memory, the communication detection and the thread mapping algorithm. These sources of overhead are depicted in Figure 14 as the percentage of the total execution time of each application. In the figure, we also show the TLB miss rate of the applications, because it determines the number of times the detection mechanism is triggered. Regarding the mapping overhead, it is lower than 1% for all the applications.

The overhead of fetching the contents of the TLB depends on the latency of the instruction that reads the TLB. We measured the latency of this instruction on a MIPS based processor (Ralink

DOI: 10.1002/cpe

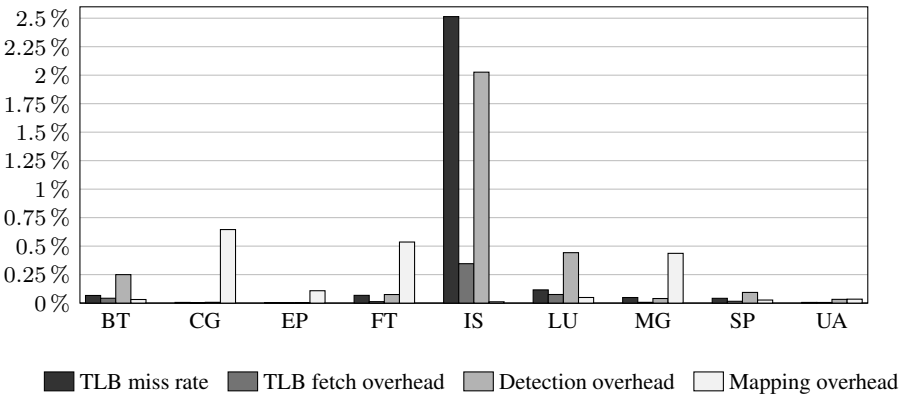


Figure 14. TLB miss rate and TLB mechanism overhead.

RT3052 MIPS 24KEc), which contains an instruction to access a TLB entry. This processor requires 4 cycles to read one entry of the TLB. In our experiments, we model a conservative value of 20 cycles to read each entry of the TLB. IS is the application with the highest overhead, because it has a high TLB miss rate and hence a high number of triggered detections. For all other applications, the TLB fetching overhead is less than 0.1%.

The overhead of our communication pattern detection mechanism depends only on the amount of TLB misses of the application, since we used the *Local TLB misses* trigger (Section 4.3). IS is also the application with the highest overhead due to the high TLB miss rate. For the other applications, the overhead of our TLB communication detection mechanism is less than 0.5% of the total execution time.

6.7. Comparison to Related Work

We compare our proposed TLB mechanism to two previous techniques: Autopin [7] and a cache miss based mechanism [10], to which we refer as *Azimi* in this section. 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 which resulted in the highest IPC was selected for the rest of the execution. Autopin was directly executed on the real machine. We implemented Azimi in the same simulated machine that was used in the main paper. The detected communication information was then fed to the real machine during execution, similar to the evaluation of our TLB mechanism.

The communication matrices obtained by Azimi are shown in Figure 15. We can observe that, for BT and LU, the detected communication pattern is incomplete, as Azimi detected the neighboring communication for few threads. For CG, EP and FT, the communication pattern was detected successfully. For the other applications, Azimi detected the wrong pattern. No communication matrix from Autopin is shown because Autopin does not detect communication, only evaluates several mappings and selects the one with highest IPC.

Figure 16 shows the average execution time of 50 executions for our proposed TLB mechanism and the related work, Azimi and Autopin. Values are normalized to the results of the operating system. Our proposal presented better improvements in all applications. Autopin, in several executions, selected a mapping different from the Oracle as best, which shows that indirect metrics do not accurately optimizes the communication. Even when it correctly selects the best mapping, its performance improvement is lower than ours because it needs to evaluate several other mappings. Regarding Azimi, the incomplete communication pattern detected by it results in sub-optimal mappings.

DOI: 10.1002/cpe

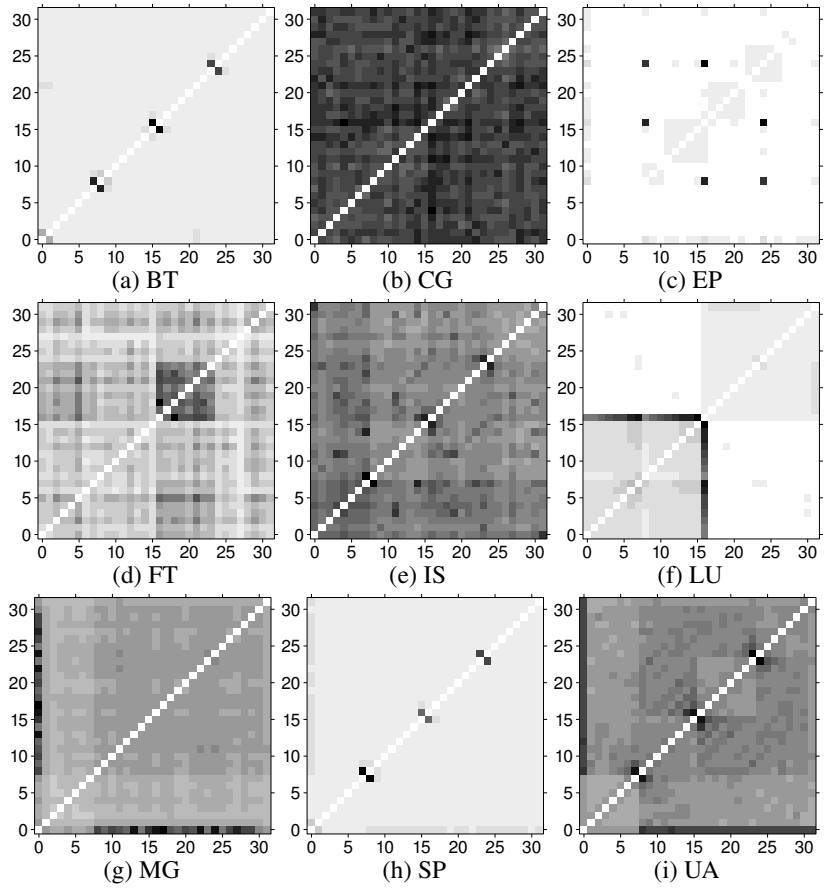


Figure 15. Overall communication patterns of the NAS benchmarks detected by the Azimi mechanism. Darker cells indicate more communication between threads.

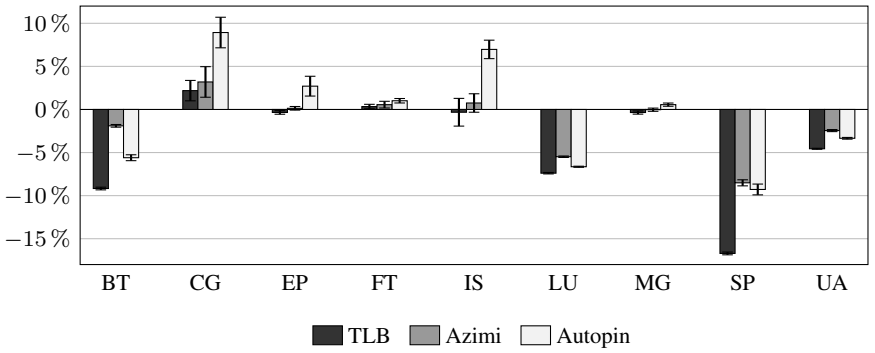


Figure 16. Application execution time when running with our proposed TLB mechanism and related work. All values are normalized to the OS.

7. CONCLUSIONS

In this paper, we presented a new mechanism to detect the communication pattern in shared memory applications using the Translation Lookaside Buffer (TLB). Our proposal detects the communication during the execution of the parallel application, allowing the operating system to perform communication-aware mapping dynamically. It presents several benefits in relation to previous approaches. It does not require changes to the source code the parallel application, which

DOI: 10.1002/cpe

makes it easier to be used, and provides a better accuracy and lower overhead. For our mechanism to work, the software must be able to read the TLB contents. In hardware-managed TLBs, this can be achieved by adding an instruction to allow the operating system to access the TLB. In software-managed TLBs, our proposal can be used without any modifications to the hardware. For these reasons, our mechanism is suitable for a wide variety of processor architectures.

We evaluated our proposal using applications from the PARSEC Benchmark Suite and the OpenMP implementation of the NAS parallel benchmarks. Our proposal was evaluated in a real machine and in a full system simulator. We were able to identify the communication patterns of all applications and used the detected communication patterns to map the threads and run performance experiments. We measured the execution time, cache misses and cache-to-cache transactions, as well as energy consumption, and compared them to other mapping strategies. The execution time was reduced by up to 45% in the simulator and 16.7% in the real machine. The total DRAM and processor energy consumptions were reduced by up to 24.3% and 14.8%, respectively. These improvements were possible because our mapping mechanism optimized the usage of cache memories and interconnections. The number of L2 and L3 cache MPKIs were reduced by up to 25.1% and 59.3%, and the number of cache-to-cache transactions by up to 76.3%.

ACKNOWLEDGEMENT

This paper was partially supported by CNPq, CAPES and FAPERGS.

REFERENCES

1. Coteus PW, Knickerbocker JU, Lam CH, Vlasov Ya. Technologies for exascale systems. *IBM Journal of Research and Development* Sep 2011; **55**(5):14:1–14:12, doi:10.1147/JRD.2011.2163967.
2. Borkar S, Chien AA. The Future of Microprocessors. *Communications of the ACM* 2011; **54**(5):67–77, doi: 10.1145/1941487.
3. Zhai J, Sheng T, He J. Efficiently Acquiring Communication Traces for Large-Scale Parallel Applications. *IEEE Transactions on Parallel and Distributed Systems* 2011; **22**(11):1862–1870.
4. Chishti Z, Powell MD, Vijaykumar TN. Optimizing Replication, Communication, and Capacity Allocation in CMPs. *ACM SIGARCH Computer Architecture News* May 2005; **33**(2):357–368, doi:10.1145/1080695.1070001.
5. Barrow-Williams N, Fensch C, Moore S. A Communication Characterisation of Splash-2 and Parsec. *IEEE International Symposium on Workload Characterization (IISWC)*, 2009, doi:10.1109/IISWC.2009.5306792.
6. Diener M, Madruga FL, Rodrigues ER, Alves MAZ, Navaux POA. Evaluating Thread Placement Based on Memory Access Patterns for Multi-core Processors. *IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2010, doi:http://doi.ieeecomputersociety.org/10.1109/HPCC.2010.114.
7. Klug T, Ott M, Weidendorfer J, Trinitis C. autopin – Automated Optimization of Thread-to-Core Pinning on Multicore Systems. *High Performance Embedded Architectures and Compilers* 2008; **3**(4):219–235.
8. Radojković P, Cakarević V, Verdú J, Pajuelo A, Cazorla FJ, Nemirovsky M, Valero M. Thread Assignment of Multithreaded Network Applications in Multicore/Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 2013; **24**(12):2513–2525.
9. Broquedis F, Aumage O, Goglin B, Thibault S, Wacrenier PA, Namyst R. Structuring the execution of OpenMP applications for multicore architectures. *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2010.
10. Azimi R, Tam DK, Soares L, Stumm M. Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring. *ACM SIGOPS Operating Systems Review* Apr 2009; **43**(2):56–65, doi: 10.1145/1531793.1531803.
11. Cruz EHM, Diener M, Navaux POA. Using the Translation Lookaside Buffer to Map Threads in Parallel Applications Based on Shared Memory. *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2012, doi:10.1109/IPDPS.2012.56.
12. Intel. 2nd Generation Intel Core Processor Family. *Technical Report September* 2012.
13. Castro M, Góes LF, Fernandes LG, Méhaut JF. Dynamic Thread Mapping Based on Machine Learning for Transactional Memory Applications. *Euro-Par Parallel Processing*, 2012; 465–476.
14. Su C, Li D, Nikolopoulos DS, Grove M, Cameron K, De Supinski BR. Critical Path-Based Thread Placement for NUMA systems. *ACM SIGMETRICS Performance Evaluation Review* 2012; **40**(2):106–112.
15. Diener M, Cruz EHM, Navaux POA. Communication-Based Mapping Using Shared Pages. *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2013, doi:10.1109/IPDPS.2013.57.
16. Cruz EHM, Diener M, Alves MAZ, Navaux POA. Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols. *Journal of Parallel and Distributed Computing* Mar 2014; **74**(3):2215–2228, doi:10.1016/j.jpdc.2013.11.006.
17. Intel. Intel 64 and IA-32 Architectures Software Developer's Manual 2013.
18. ARM. *ARM Architecture Reference Manual*. 2005.
19. Weaver DL, Germond T. *The SPARC Architecture Manual, Version 9*. 2000.

20. MIPS. *MIPS R10000 Microprocessor User's Manual, Version 2.0*. 1996.
21. Bokhari S. On the Mapping Problem. *IEEE Transactions on Computers* 1981; **C-30**(3):207–214.
22. Pellegrini F. Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. *Scalable High-Performance Computing Conference (SHPCC)*, 1994; 486–493.
23. Devine KD, Boman EG, Heaphy RT, Bisseling RH, Catalyurek UV. Parallel hypergraph partitioning for scientific computing. *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2006, doi:10.1109/IPDPS.2006.1639359.
24. Jeannot E, Mercier G. Near-optimal placement of MPI processes on hierarchical NUMA architectures. *Euro-Par Parallel Processing*, 2010.
25. Osiakwan C, Akl S. The Maximum Weight Perfect Matching Problem for Complete Weighted Graphs is in PC. *IEEE Symposium on Parallel and Distributed Processing (SPDP)*, 1990; 880–887, doi:10.1109/SPDP.1990.143503.
26. Martin M, Sorin D, Beckmann B, Marty M, Xu M, Alameldeen A, Moore K, Hill M, Wood D. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News* 2005; **33**(4):92–99.
27. Thoziyoor S, Muralimanohar N, Ahn JH, Jouppi NP, Alto P. Cacti 5.1. *Technical Report* 2008.
28. Broquedis F, Clet-Ortega J, Moreaud S, Furmento N, Goglin B, Mercier G, Thibault S, Namyst R. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. *Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010; 180–186, doi:10.1109/PDP.2010.67.
29. Jin H, Frumkin M, Yan J. The OpenMP implementation of NAS Parallel Benchmarks and Its Performance 1999.
30. Bienia C, Kumar S, Singh JP, Li K. The PARSEC Benchmark Suite: Characterization and Architectural Implications. *Parallel Architectures and Compilation Techniques (PACT)*, 2008.
31. Intel. Intel Performance Counter Monitor - A better way to measure CPU utilization 2012. URL http://www.intel.com/software/pcm.
32. Intel. Intel vtune software December 2012.
33. Torrellas J. Architectures for extreme-scale computing. *IEEE Computer* 2009; **42**(11):28–35.