2012 IEEE 26th International Parallel and Distributed Processing Symposium

Using the Translation Lookaside Buffer to Map Threads in Parallel Applications Based on Shared Memory

Eduardo H. M. Cruz, Matthias Diener, Philippe O. A. Navaux
Informatics Institute
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
{ehmcruz, mdiener, navaux}@inf.ufrgs.br

***Abstract*—The communication latency between the cores in multiprocessor architectures differs depending on the memory hierarchy and the interconnections. With the increase of the number of cores per chip and the number of threads per core, this difference between the communication latencies is increasing. Therefore, it is important to map the threads of parallel applications taking into account the communication between them. In parallel applications based on the shared memory paradigm, the communication is implicit and occurs through accesses to shared variables. For this reason, it is difficult to detect the communication pattern between the threads. Traditional approaches use simulation to monitor the memory accesses performed by the application, requiring modifications to the source code and drastically increasing the overhead.**

In this paper, we introduce a new light-weight mechanism to detect the communication pattern of threads using the Translation Lookaside Buffer (TLB). Our mechanism relies entirely on hardware features, which makes the thread mapping transparent to the programmer and allows it to be performed dynamically by the operating system. Moreover, no time consuming task, such as simulation, is required.

We evaluated our mechanism with the NAS Parallel Benchmarks (NPB) and achieved an accurate representation of the communication patterns. Using the detected communication patterns, we generated thread mappings using a heuristic method based on the Edmonds graph matching algorithm. Running the applications with these mappings resulted in performance improvements of up to 15.3%, reducing the number of cache misses by up to 31.1%.

***Keywords*-Thread mapping; Parallel applications; Shared memory; TLB; Translation Lookaside Buffer; Interconnections; Cache Misses;**

I. Introduction

The limit of instruction level parallelism is being reached, therefore the industry relies on parallelism to continue increasing the performance of processors. The increase of the number of cores aggravates the memory wall problem, as more bandwidth between the cores and the main memory is required. Memory hierarchies with several levels of cache memories are employed to overcome this issue. Some levels of the memory hierarchy are shared by more than one core, which causes the communication latency among the cores to be different. As parallel applications are becoming the standard to take advantage of multi-core architectures, it is important to consider the communication between the threads when mapping them to cores.

The communication between the threads of parallel applications based on the shared memory paradigm is implicit. Communication happens every time a thread reads or writes data that has been previously accessed by other threads, thereby replicating data in the caches. Replications of cache lines reduce the effective size of caches [1]. The objective of using communication patterns to map threads is to improve performance by mapping threads that communicate a lot to nearby cores on the memory hierarchy. This way, there is less replication of data in different caches. The caches can be used more efficiently, and the number of cache misses is reduced. Furthermore, replications impose a high overhead on cache coherence protocols. By optimizing the mapping, the numbers of cache-to-cache and invalidation transactions are decreased.

In this paper, we propose a new light-weight mechanism to find the communication pattern of parallel applications based on shared memory. Our approach consists of looking at the most recently accessed pages by each core. This was done by checking the content of the *Translation Lookaside Buffer (TLB)*, which is responsible to perform 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 all TLBs for matching entries. We developed methods for both software-managed and hardware-managed TLBs, covering most of the current architectures. As a result, our mechanism allows the thread mapping to be performed dynamically by the operating system, and does not require simulation or any changes to the source code of the applications.

The proposed mechanism was evaluated in the Simics simulator [2]. A mapping algorithm used the communication patterns obtained with our mechanism to map the threads on the cores of the machine. The mapping problem is known to be NP-Hard [3]. To avoid an exponential time complexity, a heuristic method was developed to map the threads on the cores. The heuristic method is based on the Edmonds graph matching algorithm [4], which provides a

1530-2075/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPS.2012.56

well suited thread mapping for a selected application. We performed experiments in Simics and on a real machine with the NAS Parallel Benchmarks (NPB), showing that our mechanism successfully detected the communication patterns and reduced execution time, cache misses and traffic on the interconnections.

The paper is organized as follows. Section II presents and analyzes related work. Section III presents theoretical observations about thread mapping based on communication and outlines properties of a good communication detection mechanism. Section IV introduces our proposed mechanism, which uses the TLB to determine the communication pattern. Section V shows how our proposal was evaluated. The results of our experiments are shown in Section VI. Finally, Section VII draws conclusions and presents future work.

II. Related Work

Static process mapping of message passing applications has been evaluated in Rodrigues et al. [5]. Groups of threads that send more data among them were mapped to nearby processors. Experiments were performed on a cluster, which imposes a high latency on remote accesses and has therefore a high potential for process mapping. Although performance gains of up to 9.16% were achieved, they were not as big as expected. Sonnek et al. [6] also worked on mapping message passing based applications. They dynamically mapped virtual machines on nodes of clusters, achieving up to 42% of performance improvement. In applications based on messaging passing, detecting the communication pattern is straightforward when compared to shared memory. It is accomplished by monitoring the origin and destination fields of each message.

Barrow-Williams, Fensch, and Moore [7] evaluate a technique to collect the communication pattern of the threads of parallel applications based on shared memory. They instrumented the Simics simulator to register all the memory accesses in files, which were analyzed to determine the memory communication pattern of the applications. The authors mention that the traces, even compressed, take a large amount of space (more than 100 gigabytes). Their mechanism also requires simulation, which is a time consuming task. The work from Bienia, Kumar and Li [8] also analyzes the communication pattern of applications, but using a dynamic binary analysis tool called Pin [9], which also demands a lot of time. The papers presented in this paragraph, although they obtain an accurate communication pattern, are not suitable for real applications, as they demand a lot of time and computational resources.

In Cruz et al. [10] and Diener et al. [11], the potential of mapping the threads of applications taking into account the communication between them was evaluated. The Simics simulator was instrumented to monitor all memory accesses and detect the communication patterns of the applications. With these patterns, they created a static thread mapping to measure the performance improvement. This method allows accurate detection of the communication patterns, as the simulator records information about all memory accesses. Despite improving performance, this approach is infeasible for real applications as it requires simulation and detects only a static communication pattern.

Azimi et al. [12] show that hardware performance counters already present in current processors may be used to dynamically map parallel applications. They schedule threads by taking into account an indirect estimate of the communication pattern based on stall cycles, cache miss counters and other hardware counters present in the Power5 processor. To decrease the overhead of the proposed mechanism, the mapping system is only enabled after the number of core stall cycles exceeds a given threshold. Performance was increased by up to 7%, and the number of memory accesses to remote cache memories was reduced by up to 70%.

Broquedis et al. [13] developed the ForestGOMP mapping library. This library integrates into the OpenMP [14] runtime environment and gathers information about the different parallel sections of applications from hardware performance counters. The library generates data and thread mappings for the regions of the application. The data mapping is suitable for Non-Uniform Memory Access (NUMA) machines, as in this machines the latency to the memory banks may be different for each processor. The library tries to keep the threads that share data nearby according to the memory hierarchy, as well as to place the memory pages in NUMA nodes close to the core that is accessing the page. They improved performance by up to 11.6% using the NAS parallel benchmarks. The main disadvantage of these two papers is that the hardware counters can only be used to estimate the communication pattern between the threads indirectly. In contrast, our approach using the TLB provides more accurate information about the communication pattern.

III. Theoretical Observations

In this section, we contextualize our work by showing the benefits of using communication to map threads in multicore systems. Furthermore, we explain the properties that a mechanism to find the communication pattern should have, as well as how to group the communication.

A. Using Communication to Map Threads

The basic goal of process and thread mapping is to optimize the usage of the available resources, thereby improving performance and minimizing energy consumption. In multicore systems, threads that communicate a lot should be mapped close to each other in the memory hierarchy.

There are two objectives in optimizing the thread mapping using the communication, reducing the number of cache misses and improving the use of interconnections.

1) Reducing Cache Misses: The most important goal is to reduce the number of cache misses. To address them properly, we have to differentiate between three types of cache misses, cache misses due to invalidations, capacity cache misses and cache misses due to replication.

Invalidation misses are cache misses that happen when the requested data is continually invalidated due to cache coherence protocols. For instance, the MESI [15] protocol sends invalidation messages every time a write is performed in shared cache lines. A common situation in shared-memory programs is to have one thread writing to an area of memory and another reading from the same area. In this case, the writer thread will successively invalidate the cache lines of the other thread. Therefore, write operations impact more on performance than read operations, as all writes to shared cache lines invalidate the corresponding lines on the other caches.

The objective is to reduce invalidation misses that happen when two or more caches hold the same data and are continuously invalidated by the other cache. It is important to note that memory accesses to data are more relevant than instruction fetches when mapping the threads. The reason is that write operations to data occur frequently, while write operations to instructions only occur when the operating system loads a program into memory.

Capacity misses are cache misses that happen when data is accessed for the first time and was not already fetched from memory. In shared-memory programs, threads that access a lot of memory and share a cache will evict cache lines from each other and cause capacity misses. The objective is to reduce competition for cache lines between cores that share a cache.

Replication misses are cache misses that happen due to uncontrolled cache line replication. A cache line is said to be replicated when it is present in more than one cache. As stated in [1], uncontrolled replication leads to a virtual reduction of the size of the caches, as some of their space would be used to store the same cache lines. By mapping applications that share large amounts of data to cores that share a cache, the space wasted with replicated cache lines is minimized, leading to a reduction of cache misses.

2) Improving the Use of Interconnections: The second objective of thread mapping is to make better use of the available interconnections in the processors. The goal is to reduce inter-chip traffic and use intra-chip interconnections instead, which have a higher bandwidth and lower latency. In order to reach this objective, the numbers of snoop transactions and cache line invalidations have to be reduced.

B. Properties of a Mechanism to Detect the Communication Pattern

To be suitable for a real-world environment, a mechanism to find the communication pattern between threads should have the following properties.

1) Detect communication pattern during execution: Many previous approaches rely on finding the communication pattern in a phase before the actual execution of the workload, for example by using simulation or binary instrumentation. This is very time-consuming and potentially takes a lot of storage space to store intermediate data, such as memory traces.

2) Low impact on performance: The mechanism should have a very low overhead in order not to interfere with execution of the application. Some previous approaches permute the mapping of threads to cores periodically to observe changes in the cache statistics. As a remapping of threads has an overhead in terms of an increase of cache misses, this leads a noticeable decrease of performance and is therefore less efficient.

3) Provide an accurate communication pattern: The detection of the communication pattern should be as accurate as possible to allow a beneficial mapping to be done. In the case of shared memory applications, this means that the observation should happen as directly as possible. Approaches that use hardware counters, for example, only observe the applications behavior indirectly and provide a less accurate view of the communication between the threads.

4) Detect dynamic behavior: Since applications change their behavior and therefore the communication pattern during the execution, the mechanism should be able to detect changes dynamically and thereby make dynamic mapping possible. Many previous approaches analyze the application over the whole execution time and provide a static mapping for the application. This leads to wrong results when the application exhibits dynamic behavior.

5) Avoid the false communication problem: Related to dynamic behavior, false communication means that threads appear to communicate through shared data, yet in reality they are not communicating. The root of the problem is that communication is implicit and happens through shared memory. As an example, false communication can happen when two threads access the same address, but at different times during the execution. Another example is the classical false sharing problem, in which a cache line is present in more than one cache, but the cores are accessing different addresses inside the cache line. Both examples should not be considered as communication.

6) Independent from the application implementation: To provide benefits to a wide number of applications, the mechanism should be transparent to the programmer and user and make as few assumptions about the applications as possible. This has two consequences. First, using the mechanism should not depend on a particular parallelization API, such as OpenMP and Pthreads. Second, it should not require the programmer to modify the source code or link to additional libraries.

As explained in Section II, many previous approaches lack one or more of these properties. We propose a new

light-weight mechanism to find the communication pattern dynamically, which has the properties listed above. We will introduce this mechanism and evaluate it according to the properties outlined here in Section IV.

C. Finding Groups of Communication

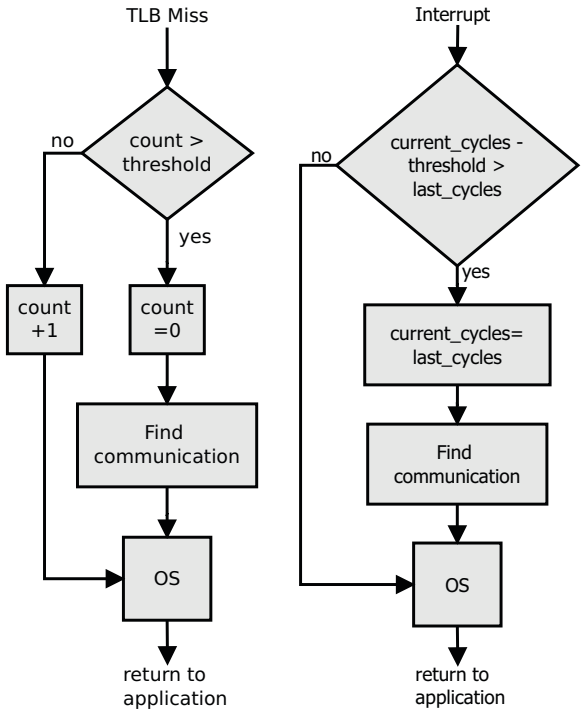
Communication can be analyzed by grouping different numbers of threads. To calculate the amount of communication between groups of threads of any size, the time and space complexity rises exponentially, which discourages the use of thread mapping for large groups of threads. Therefore, the communication was evaluated only between pairs of threads, thus generating a *communication matrix*. Although this can decrease the accuracy of the results, it reduces the complexity to $\Theta(N^2)$, where N is the number of threads, and allows a faster processing of the information.

IV. A New Mechanism Using the TLB to Detect the Communication Pattern of Shared Memory Applications

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 make this translation possible. These *page tables* contain the physical address for each virtual page and are indexed by the higher bits of the virtual address for quick translation. However, the memory accesses to the page table impose a high overhead, and some architectures even require several accesses, in case the page table consists of more than one level. To overcome these issues, 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.

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 and register every time two entries match, we get the amount of pages shared by the cores as a result. By systematically doing this procedure, we get a representation of the communication pattern at the page level granularity. This pattern can be used to map the threads of the applications on modern multi-core architectures, taking advantage of shared cache memories and intra-chip communication.

Current processor architectures manage TLBs in different ways. The two most important types of management, software-managed and hardware-managed TLBs, require slightly different methods to detect the communication pattern. Therefore, we describe our proposed mechanism separately for each of the two TLB management types.



(a) Software-managed TLB. (b) Hardware-managed TLB.

Figure 1. Flowcharts for the proposed mechanism.

A. Software-Managed TLBs

In some RISC architectures, such as SPARC [16], 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. The main advantages of this management type are that it simplifies the hardware and is very flexible, since the operating system can choose how to implement the virtual memory.

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 all the other TLBs for matches besides loading the entry from the main memory. Accessing other TLBs could represent a bottleneck. To overcome this issue, the contents of all TLBs can be mirrored in the main memory. This would not require much storage space, as the size of the TLB is usually small to keep access latency low. To further reduce the impact of iterating over the TLBs, the operating system could treat the TLB as a set associative cache, so that only a few entries from each TLB would have to be compared for matches. Also, instead of running the search for every TLB miss, the search could be run for only a fraction of them. This decreases the

accuracy but reduces the overhead to the same fraction used for sampling.

Our implementation of this proposal for the software-managed TLB is presented in the flowchart in Figure 1(a). When a TLB miss occurs, we compare a counter against a previously selected threshold. If the counter is below the threshold, we just increment it and return to the operating system to reduce the overhead imposed by our mechanism. Otherwise, we set the count to zero and search for the requested address in the other TLBs in the system, incrementing the communication matrix whenever a match is found. Finally, we return to the operating system, which fetches the TLB entry and returns control to the application.

The time complexity to find the communication in a fully associative software-managed TLB is $\Theta(P \cdot S)$, where P is the number of processing cores and S is the size of the TLB. The complexity increases linearly with P since we have to check all the other TLBs, it is also linear with S because all the entries of the TLB are searched for matches. However, considering a set associative TLB, the time complexity decreases to $\Theta(P)$, because the associativity is a constant and is much smaller than the size of the TLB. In this case, it is not necessary to check all the entries of the TLB for matches, but just the entries that are on the same set.

B. Hardware-Managed TLBs

Architectures such as x86 and x86-64 [17] use the TLB only as a cache for the page table entries stored in the main memory. 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 main memory and loads the corresponding entry into the TLB. This mechanism is called hardware-managed TLB. The operating system only keeps the content of the page table in the main memory. The only management that is performed by the operating system in this type of TLB is the invalidation of entries when the page table is modified. The hardware-managed approach has a low impact on performance, as it does not require traps and context switches on every TLB miss.

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 enables the operating system to access the content of the TLB. This way, the kernel could search the TLBs for matching entries periodically. Accuracy and overhead of this mechanism depend on the time between searches. To increase the accuracy and reduce the overhead of this mechanism, an optional modification could be used to signal the kernel when a TLB miss occurs.

Our implementation of this proposal for the hardware-managed TLB is presented in the flowchart in Figure 1(b). Whenever an interrupt occurs, we subtract a previously defined threshold from the cycle counter and compare it to the value of the cycle counter when the last search occurred. The reason for this comparison is to limit the number of times a search for communication runs, and thereby decrease the overhead of the mechanism. If not enough time passed since the last search, we just return to the operating system. Otherwise, we store the current value of the cycle counter and proceed to search all TLBs for matching addresses, incrementing the communication matrix for each match. Finally, we return to the operating system.

If the hardware-managed TLB is fully associative, the time complexity for the algorithm to find the communication is $\Theta(P^2S^2)$, where P is the number of processing cores and S is the size of the TLB. The complexity is quadratic in P 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 S. However, if we use a set associative hardware-managed TLB, the time complexity is decreased to $\Theta(P^2S)$ as it is not necessary to check all the entries of the TLB for matches, but just the entries that are on the same set.

Table I
COMPARISON OF THE PROPOSED MECHANISM FOR SOFTWARE-MANAGED AND HARDWARE-MANAGED TLBS.

	Software-managed TLB	Hardware-managed TLB
Example Architecture	Sparc, MIPS	Intel
Trigger mechanism for finding communication	triggered every n TLB Misses	triggered every n million cycles
Value for n used in our experiments	100	10,000,000
Search which TLBs for matches	all pairs with TLB where miss occurred	all possible pairs of TLBs
Time complexity for set associative TLB	$\Theta(P)$	$\Theta(P^2S)$
Hardware modification necessary?	No	Yes

C. Characteristics of the Proposed Mechanism

A summary of the properties of the methods for the software-managed and hardware-managed TLBs is shown in Table I. As outlined in Section III-B, a mechanism to detect communication patterns between threads should have several properties.

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 time of the application. Regarding the dynamic behavior of applications, 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 after a short time. It will therefore not be counted anymore in the calculation of the communication pattern. Similarly, the impact of false communication is greatly reduced by the relatively short life of the TLB entries. Regarding the classical false sharing problem, any access to the same memory page is considered as communication, regardless of the offset.

The complexity of the mechanism is lower for the software-managed TLB than for the hardware-managed TLB. In the mechanism for the software-managed TLB, we know that the communication pattern has only changed for the core where the TLB miss happened, and we search for communication only between this core and the others. In the mechanism for the hardware-managed TLB, we do not have this information and have to search for communication in all possible pairs of TLBs. We evaluate the accuracy and the overhead of the mechanism in Sections VI-A and VI-C, respectively.

V. Evaluation of our proposal

This section explains how we evaluated our proposed mechanism. We execute the NAS benchmarks inside a simulator to detect the communication patterns. To evaluate the performance, we use the detected communication pattern to create a static mapping from threads to cores for each benchmark and execute them on a real machine. The number of threads is equal to the number of cores, and each thread gets mapped to a different core. We do not migrate the threads during the execution of the applications. Dynamic migration requires an algorithm to detect when the communication pattern changes [18], as well as substantial modifications to the scheduler of the operating system. These are beyond the scope of this work, which is to present a mechanism that dynamically detects the communication patterns.

The rest of this section is organized as follows. First, we give a short overview of the algorithm we used to map threads to cores. This algorithm uses the communication pattern of an application as detected by our mechanism. Then, we present the simulation environment we used and outline the parameters of the simulated machines. Finally, we talk briefly about the workloads we used to evaluate our proposal.

A. Creating the Thread Mapping

After generating the communication matrix, it is necessary to map the threads to the cores. The mapping problem is considered NP-Hard [3], consequently, finding the optimal solution becomes infeasible when the number of threads grows. Thus, heuristic algorithms must be employed to determine the mapping in a reasonable time, with results as close as possible to the optimal mapping. Methods such as *Dual Recursive Bipartitioning* produce good results, and

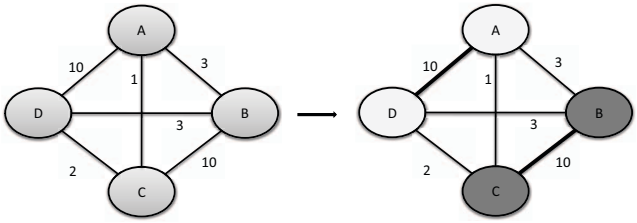


Figure 2. The Matching Problem.

are available in the Scotch [19] library. However, in this work, a different method was used to obtain the mapping, based on the maximum weight perfect matching problem for complete weighted graphs, as presented in [10].

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 output of the algorithm can be seen in Figure 2. This problem can be solved by the Edmonds graph matching algorithm in polynomial time [4]. A parallel version of this 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 number and P is the number of processors.

To model thread mapping as a matching problem, the vertices represent the threads and the edges the amount of communication. A complete graph is obtained directly from the communication matrix. The graph is processed by the matching algorithm, which outputs the pairs of threads so that the amount of communication is maximized.

In many processor architectures, there are only two cores sharing the a cache, therefore, mapping threads to them with the matching algorithm is straightforward. However, there are architectures where more than two cores share the same cache, or there are more levels of memory hierarchy to be exploited. In these cases, the matching algorithm by itself is insufficient. To overcome this issue, another communication matrix, containing the communication between pairs of pairs of threads, is given as input and the algorithm is re-executed. This matrix was generated by the following heuristic function:

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

where (x, y) and (z, k) are the matches found in the previous step, and $M_{(i,j)}$ is the amount of communication between threads i and j. This algorithm does not guarantee that the result will contain the pairs of pairs with the most amount of communication, as the communication matrix does not provide sharing information about groups with more than 2 threads. However, it is a reasonable approximation and keeps the time and space complexity polynomial.

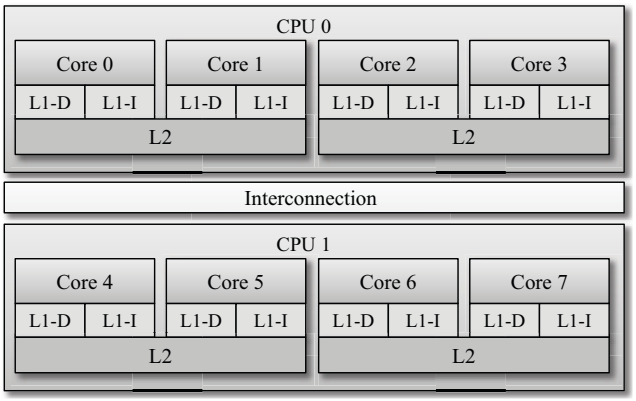


Figure 3. Architecture used to evaluate our proposal.

B. Hardware Environment

We simulated our proposed mechanism in the Simics [2] full system simulator. The system architecture we used in our simulations is presented in Figure 3. The system contains two physical processors, modeled after the Intel Harpertown architecture [20], each consisting of four cores. Each core has private L1 caches for data and instructions, but the L2 cache is shared between two cores. The memory latencies were calculated using the CACTI tool [21]. The configuration of the caches is shown in Table II. The real machine used for performance evaluation consists of two Intel Xeon E5405 processors, based as well on Harpertown, which have the same memory hierarchy as the simulated machine.

C. Workloads

In the rest of the paper, we used the OpenMP implementation of the NAS parallel benchmarks (NPB) [22] to evaluate our proposal. The benchmarks were executed using the *W* input size, since it is the most appropriate size for simulation. We ran all the benchmarks except DC, which takes too much time to simulate.

VI. Results

In this section, we present the results we obtained by using our proposed mechanism on the benchmarks. First, we show the communication patterns we detected. Afterwards, we detail the performance improvements we made by mapping the threads using the communication patterns. From now on, we will call the approach for the software-managed TLB *SM*, and the approach for hardware-managed TLB *HM*. All results presented in this section are average values obtained by executing each benchmark 100 times.

A. Communication Patterns

Figures 4 and 5 show the communication patterns of the NPB applications for SM and HM, respectively. Each cell *(i,j)* represents the communication between threads i and j. Darker cells signify a higher amount of communication. The size of the TLBs was 64 entries for both approaches, with a 4-way set associativity. This is the default size of the TLB in UltraSparc, as well as the size of the L1 TLB in the Intel Nehalem architecture. For SM, only in 1% of the TLB misses a search for matches was performed. We also simulated SM monitoring all the TLB misses, but we chose to present the patterns with only 1% of the samples due to the overhead issues mentioned before. HM was evaluated with 10 million cycles between each search for matches.

Table II
Configuration of the caches.

Parameter	L1 Cache	L2 Cache
Size	32 KBytes	6 MBytes
Number	8 Inst. + 8 Data	4 (shared by 2 cores)
Line Size	64 Bytes	64 Bytes
Set Associativity	4 Ways	8 Ways
Latency	2 Cycles	8 Cycles
Protocol	Write-through	Write-back, MESI

In general, the communication pattern detected by SM is more accurate. The reason is that SM is able to access more samples than HM, as all the TLB misses are handled by the operating system. BT is an application that presents a lot of communication between neighboring threads. This is very common when the application is based on domain decomposition, where most of the shared data is located on the borders of each sub-domain. For BT, both SM and HM were able to detect the communication pattern. One difference is that HM detected more communication between thread 7 and all other threads than SM. As a result, the mapping algorithm found an optimal mapping for SM, and a slightly worse mapping for HM. The domain decomposition pattern is also present in IS, LU, MG, SP and UA. Additionally, LU also presents communication with the most distant threads [10].

The SM approach succeeded in the IS application. In MG, it managed to detect that thread pairs 4-5 and 6-7 present more communication among them compared to thread pairs 0-1 and 2-3. Nevertheless, our mapping algorithm was able to map the threads correctly, because, as similar the patterns of threads 0 to 3 are, there is still a difference. When we took all TLB misses into account with SM, the generated pattern clearly identified the communication pattern of MG.

In IS and MG, HM detected a large amount of communication between two threads and all the other ones. The reason for this result is the runtime behavior of the applications, which can present a challenge to HM. 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

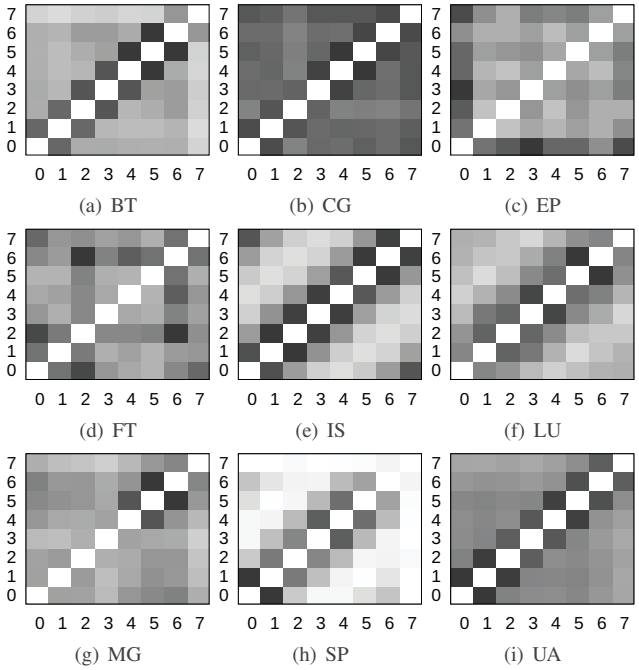


Figure 4. Communication patterns of the applications from NPB detected by the software-managed TLB mechanism.

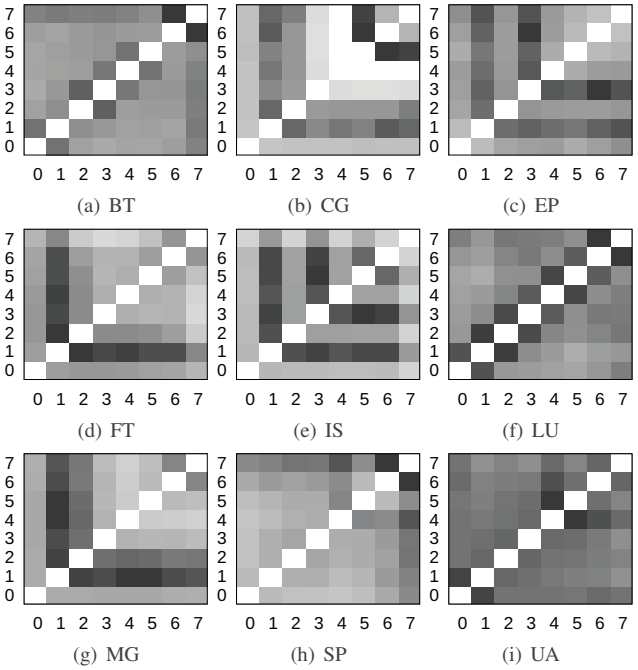


Figure 5. Communication patterns of the applications from NPB detected by the hardware-managed TLB mechanism.

may not characterize the global behavior of the application. If this happens several times, HM 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.

In LU, both SM and HM detected the domain decomposition pattern, but only SM detected that there is also a considerable amount of communication between more distant threads. For SP and UA, only the SM mechanism was able to identify the communication patterns accurately. HM, as explained before, suffers from unpredictable runtime behavior and bad samples. However, the results obtained by HM with SP and UA are much better than the results with IS and MG, as none of the threads presented a large amount of communication to all others. Furthermore, in UA, even though the domain decomposition pattern obtained with HM is not as evident as with SM, our mapping algorithm was able to find the optimal mapping.

The applications CG, EP and FT present homogeneous communication patterns [10]. Homogeneous means that their communication patterns are expected to approximately present the same amount of communication among the threads. CG, with the SM mechanism, also shows traces of a domain decomposition pattern. Nevertheless, it is notable that the proportion of the memory shared by the neighbors in CG is less expressive compared to BT, IS, LU, SP and UA. CG, EP and FT presented more homogeneous patterns with SM than HM, which reinforces the notion that runtime behavior is a key point to be considered in HM.

B. Evaluating the Performance using Thread Mapping

The communication patterns were used by our thread mapping algorithm to map the threads on cores according to the amount of communication and executed on the real machine described in Section V-B.

Figure 6 presents the execution time of the NPB applications normalized to the time of the original scheduler of the operating system, denoted by OS. To get a more detailed understanding of the benefits of thread mapping, we also monitored the number of cache line invalidations, snoop transactions and L2 cache misses. Snoop transactions happen when a core requests data that is not present in its cache and has to retrieve the data from another cache. We focused only on the L2 cache misses because the L1 caches are private and do not benefit from mapping. Figures 7, 8 and 9 show the normalized numbers of invalidations, snoop transactions and L2 cache misses, respectively. Table IV contains the values for number of invalidations, snoop transactions and cache misses, all divided by the execution time. Table V shows the standard deviations.

The number of invalidations, snoop transactions and cache misses in BT were reduced significantly. The execution time also was reduced, but by a small factor. Except for cache misses, the results with SM are a little better than HM, as the mapping found for SM was better. HM obtained slightly less cache misses, but this difference is negligible due to the high

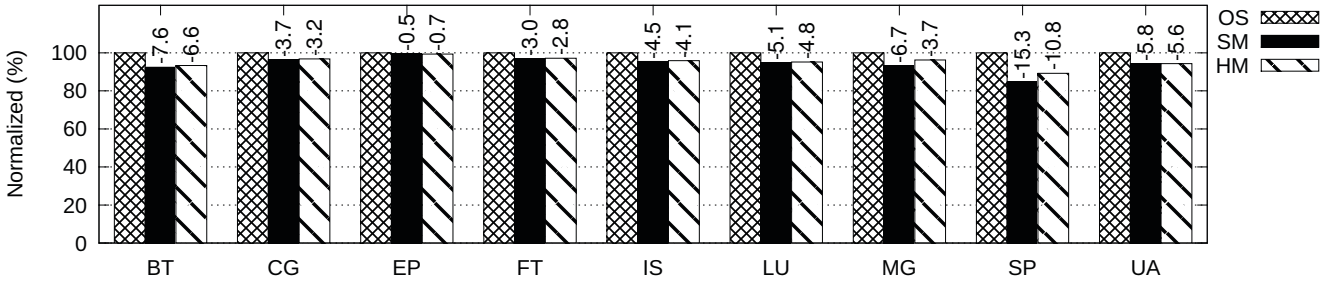


Figure 6. Execution time of the applications.

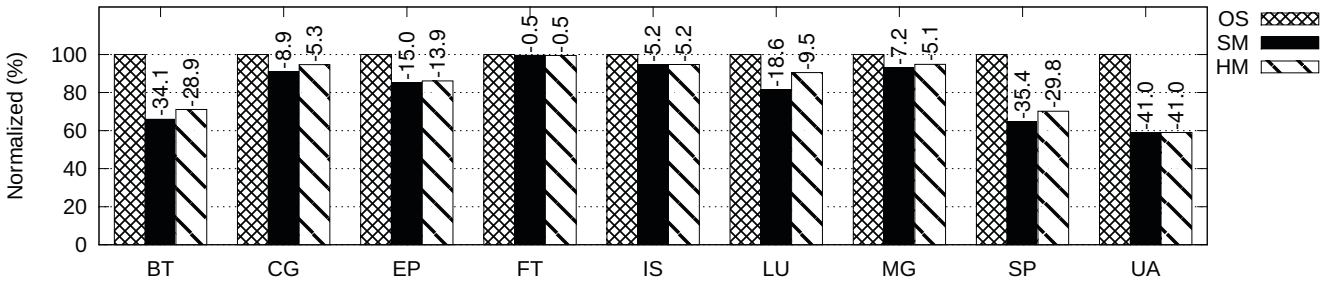


Figure 7. Invalidations due to the cache coherence protocol.

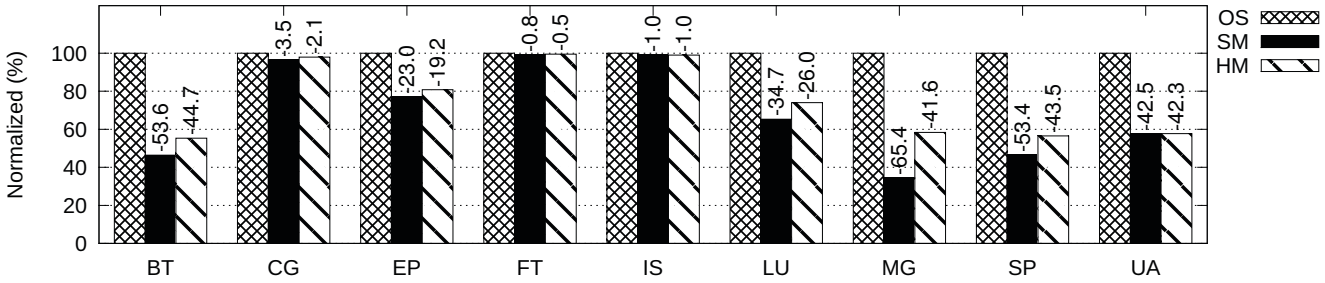


Figure 8. Snoop transactions.

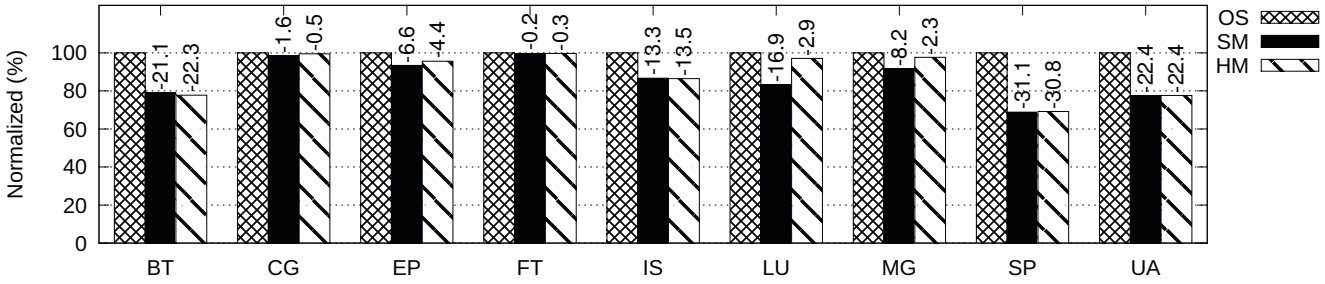


Figure 9. L2 cache misses.

standard deviation. Furthermore, the number of invalidations and snoop transactions is much more sensitive to thread mapping than cache misses. The reason is that a better mapping directly influences the number of invalidations and snoop 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. Likewise, the number of invalidations and snoop transactions is also more sensitive to thread mapping than the execution time.

SP presented the best results for execution time and number of cache misses, reducing them by 15.3% and 31.1%, respectively. SM performed better than HM in execution time, number of invalidations and snoop transactions, as only SM detected the correct communication pattern. For cache misses, the results were similar for SM and HM, but with high standard deviations. UA achieved the highest reduction of the number of invalidations (41%). The improvements in UA were virtually the same for SM and HM, since they both found the optimal thread mapping.

In LU, the performance of SM and HM was almost the same. However, as SM detected that LU also presents communication between distant threads, it significantly reduced the amount of invalidations and snoop transactions compared to HM. Although the number of cache misses was also reduced compared to HM, the standard deviation is high. It is important to note that, for the execution time, we also reduced the standard deviation. This is important because it indicates that the operating system scheduler maps the threads incorrectly during many executions.

MG is the benchmark that presented the highest reduction of the number of snoop transactions (65.4%). SM performed better than HM, as the detected communication pattern was more accurate. By looking at the absolute values of invalidations and snoop transactions for BT, SP, UA, LU and MG, we see that the proportion between invalidations and snoop transactions in MG is much lower compared to the others. Additionally, among the benchmarks just mentioned, MG was the benchmark that presented the lowest reduction of the number of invalidations. This is the reason that MG presented lower reductions of cache misses, despite having the best relative reduction of snoop transactions.

As stated in Section III, thread mapping improves performance by mapping threads that share data to cores which are close in the memory hierarchy. If the communication pattern among the threads is homogeneous, no performance improvement can be achieved by thread mapping, as in the case of CG, EP and FT. CG and FT also have short execution times, which makes them more vulnerable to unpredictable behavior during execution, resulting in high standard deviations for the execution time. In CG, the domain decomposition pattern detected by SM, despite being very subtle, was able to reduce the number of invalidations slightly compared to HM. For the other measures of CG, the standard deviations make the small improvements negligible.

Table III
STATISTICS FOR THE SOFTWARE-MANAGED TLB.

App.	TLB Miss Rate	TLB Misses for which we run SM	Total Overhead
BT	0.01%	0.655%	0.195%
CG	0.015%	0.942%	0.249%
EP	0.002%	0.998%	0.027%
FT	0.007%	0.961%	0.12%
IS	0.333%	0.993%	4.077%
LU	0.026%	0.875%	0.519%
MG	0.008%	0.82%	0.117%
SP	0.032%	0.909%	0.751%
UA	0.005%	0.829%	0.08%

EP, besides having a homogeneous communication pattern, does not share data between the threads, which is the reason that the absolute values for number of invalidations and snoop transaction are low compared to the other applications. These low values imply that small, unpredictable changes in the runtime behavior drastically alter the statistics, as can be seen in Figures 7 and 8. Additionally, by looking at the standard deviations of EP, we see that they surpass the improvements, showing that no improvements can be achieved by thread mapping in EP, which is the expected result.

Regarding IS, the communication matrix is not homogeneous. However, the execution time of IS is very low and, as stated before, the results are influenced by unpredictable behavior during execution, leading to a high standard deviation for the time measured.

C. Overhead of the Mechanism

To decrease the overhead of our proposed mechanism, we used 4-way set associative software-managed and hardware-managed TLBs. As stated in Section IV, this reduces the time complexity of the algorithms that detect the communication.

Table III presents statistics for SM. It shows the TLB miss rate, the percentage of TLB misses for which we run SM, and the total overhead of the mechanism. Since we use sampling, we only executed the search for matches for 1% of the TLB misses. The real sampling rate is even lower than that, since we are only interested in TLB misses due to data accesses, as explained in Section III.

The communication detection routine of SM takes only 231 cycles to execute. Due to this low value, all applications except IS presented a very low overhead of less than 1% for SM. Since the overhead is directly proportional to the amount of TLB Misses and IS has more than 10 times the number of TLB misses compared to the other applications, it is expected that IS would present the highest overhead. The second highest overhead is from SP, with only 0.75% of overhead. The lowest overhead was in EP (0.027%).

Table IV
EXECUTION TIME AND NUMBER OF INVALIDATIONS, SNOOP TRANSACTIONS AND L2 CACHE MISSES PER SECOND.

Parameter	Mapping	BT	CG	EP	FT	IS	LU	MG	SP	UA
Execution Time (seconds)	OS	0.74	0.13	0.48	0.10	0.06	2.39	0.23	2.53	2.19
	SM	0.68	0.13	0.47	0.10	0.06	2.27	0.22	2.14	2.06
	HM	0.69	0.13	0.47	0.10	0.06	2.27	0.22	2.25	2.06
Invalidations / second	OS	9,845,216	3,831,746	121,230	16,154,353	9,754,232	14,457,991	35,970,058	17,749,230	7,361,187
	SM	7,019,908	3,624,698	103,558	16,571,898	9,681,120	12,395,757	35,792,412	13,535,357	4,609,197
	HM	7,499,308	3,747,079	105,117	16,544,292	9,637,287	13,745,080	35,439,765	13,956,912	4,600,673
Snoop Transactions / second	OS	7,196,937	10,374,266	27,870	5,172,957	11,461,581	12,706,165	4,093,348	10,668,132	5,008,487
	SM	3,612,138	10,395,271	21,560	5,288,628	11,889,910	8,739,948	1,519,446	5,874,685	3,055,559
	HM	4,263,300	10,492,865	22,666	5,298,599	11,830,896	9,881,274	2,482,490	6,757,793	3,064,284
L2 Misses / second	OS	248,962	1,144,400	3,365	460,250	1,007,312	656734	939,658	339,850	741,887
	SM	212,403	1,169,066	3,159	473,133	914,644	575,242	924,153	276,327	610,845
	HM	207,314	1,176,111	3,240	472,221	908,205	669,864	953,271	263,512	610,188

Table V
STANDARD DEVIATIONS FOR THE PERFORMANCE EXPERIMENTS.

Parameter	Mapping	BT	CG	EP	FT	IS	LU	MG	SP	UA
Execution Time	OS	3.44%	11.35%	5.13%	20.55%	21.26%	6.98%	9.22%	1.35%	1.76%
	SM	4.15%	2.68%	1.98%	6.83%	4.62%	0.2%	2.82%	0.11%	0.25%
	HM	0.79%	4.62%	1.87%	6.13%	11.11%	1.17%	3.11%	0.11%	1.21%
Invalidations	OS	4.68%	1.45%	30.68%	0.88%	1.52%	4.55%	1.64%	4.75%	1.92%
	SM	3.41%	0.92%	22.79%	0.58%	0.68%	0.16%	2.22%	0.42%	0.97%
	HM	5.69%	1.37%	18.89%	0.48%	0.86%	1.29%	1.95%	8.36%	1.3%
Snoop Transactions	OS	5.08%	1%	32.53%	1.02%	0.78%	8.45%	7.75%	8.35%	5.79%
	SM	5.72%	0.47%	52.32%	0.73%	0.81%	1.21%	12.03%	1.29%	3.56%
	HM	6.34%	1.13%	44.21%	1.4%	1.01%	2.6%	1.03%	4.6%	3.36%
L2 Misses	OS	25.74%	1.92%	41.1%	5.28%	2.75%	11.32%	4.6%	30.04%	8%
	SM	23.89%	2.37%	38.4%	5.18%	3.3%	26.41%	4.96%	36.94%	15.03%
	HM	22.82%	2.98%	32.14%	5.25%	3.3%	14.94%	7.47%	37.48%	15.12%

The hardware-managed TLB causes the same overhead for all applications. The reason is that sampling is done with a fixed frequency, in contrast to SM, which depends on the number of TLB misses the application causes. The communication detection routine for HM requires 84297 cycles to execute. This large difference between the number of SM and HM cycles can be explained by the time complexity that we calculated in Section IV. As we performed the sampling only every 10 million cycles, the overhead of HM is less than 0.85% for our experiments.

VII. CONCLUSIONS AND FUTURE WORK

With the increase of the number of cores per chip, taking communication into account when mapping threads to cores is becoming more important. In this paper, we presented a new method that uses the TLB to find the communication patterns between threads in shared-memory applications. Our mechanism relies on hardware features and allows the communication patterns to be detected dynamically by the operating system during the execution of the application. Furthermore, it is independent of the implementation of the parallel application. We developed methods for both software-managed and hardware-managed TLBs, covering most of the current processor architectures. In contrast to traditional approaches, it does not require time consuming tasks such as simulation or modifications to the source code of the applications. Additionally, our mechanism presented a very low overhead for both TLB types. For the software-managed TLB, our method can be used without any modifications to the hardware. For the hardware-managed TLB, the only modification required is the addition of an instruction to allow the operating system to access the TLB. For these reasons, our mechanism is suitable in real-world scenarios.

We evaluated our proposal using applications from the NPB. We were able to identify the communication patterns for all NPB applications and used the detected communication patterns to map the threads and run performance experiments. We measured the execution time, number of cache line invalidations, snoop transactions and cache misses and compared them to the operating system scheduler. Performance was improved by up to 15.3%, and the number of cache misses was reduced by up to 31.1%. Cache line invalidations and snoop transactions were reduced by up to 41% and 65.4%, respectively. This shows that thread

mapping allows a much better use of the interconnections of current architectures.

In all benchmarks, performance was improved compared to the operating system scheduler. Additionally, the variability of the results was reduced in most experiments, making the application performance more predictable. 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 greatest improvements. Applications that have homogeneous communication patterns did not present improvements. This is the expected result, as there is no difference in the communication among the threads to be exploited. Expected performance improvements in NUMA architectures are higher, because of larger differences in communication latencies.

For the future, we intend to improve the approach for the hardware-managed TLB and develop dynamic migration strategies which use the mechanisms described here.

ACKNOWLEDGMENT

This work was partially supported by CNPq and CAPES.

REFERENCES

[1] Z. Chishti, M. Powell, and T. Vijaykumar, "Optimizing replication, communication, and capacity allocation in CMPs," in *International Symposium on Computer Architecture, (ISCA 2005)*.

[2] 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*, no. 2, pp. 50–58.

[3] R. Koppler, "Geometry-Aided Rectilinear Partitioning of Unstructured Meshes," in *Parallel Computation*, pp. 450–459.

[4] C. Osiakwan and S. Akl, "The maximum weight perfect matching problem for complete weighted graphs is in pc," in *Parallel and Distributed Processing, 1990. Proceedings of the Second IEEE Symposium on*, 9-13 1990, pp. 880–887.

[5] E. Rodrigues, F. Madruga, P. Navaux, and J. Panetta, "Multi-core aware process mapping and its impact on communication overhead of parallel applications," in *IEEE Symposium on Computers and Communications (ISCC 2009)*.

[6] J. Sonnek, J. Greensky, R. Reutiman, and A. Chandra, "Starling: Minimizing communication overhead in virtualized computing platforms using decentralized affinity-aware migration," in *Parallel Processing (ICPP), 2010 39th International Conference on*, Sept. 2010.

[7] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterisation of Splash-2 and Parsec," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct. 2009.

[8] C. Bienia, S. Kumar, and K. Li, "Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, Sept. 2008.

[9] M. 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," *Computer*, vol. 43, no. 3, pp. 34–41, 2010.

[10] E. Cruz, M. Alves, A. Carissimi, P. Navaux, C. Ribeiro, and J. Mehaut, "Using memory access traces to map threads and data on hierarchical multi-core platforms," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, may 2011, pp. 551 –558.

[11] M. Diener, F. Madruga, E. Rodrigues, M. Alves, J. Schneider, P. Navaux, and H.-U. Heiss, "Evaluating thread placement based on memory access patterns for multi-core processors," in *International Conference on High Performance Computing and Communications (HPCC 2010)*.

[12] R. Azimi, D. K. Tam, L. Soares, and M. Stumm, "Enhancing operating system support for multicore processors by using hardware performance monitoring," *SIGOPS Oper. Syst. Rev.*, vol. 43, 2009.

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

[14] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, no. 1, pp. 46–55.

[15] L. Ivanov and R. Nunna, "Modeling and Verification of Cache Coherence Protocols," in *International Symposium on Circuits and Systems (ISCAS 2001)*, vol. 5, 2001, pp. 129–132.

[16] SPARC, *The SPARC Architecture Manual Version 9*, SPARC International, Inc., September 2000.

[17] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Volumes 1, 2 and 3: Basic Architecture, Instruction Set Reference and System Programming Guide*, Intel Corporation.

[18] C. Ma, Y. M. Teo, V. March, N. Xiong, I. Pop, Y. X. He, and S. See, "An approach for matching communication patterns in parallel applications," in *International Symposium on Parallel Distributed Processing (IPDPS 2009)*.

[19] F. Pellegrini, "SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package."

[20] Intel, "Quad-Core Intel ® Xeon ® Processor 5400 Series Datasheet," Tech. Rep. March.

[21] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi, "A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08, 2008, pp. 51–62.

[22] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS Parallel Benchmarks," *International Journal of High Performance Computing Applications*, no. 3, pp. 66–73.