2013 IEEE 27th International Symposium on Parallel & Distributed Processing

Communication-Based Mapping Using Shared Pages

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

***Abstract*—In current shared memory architectures, the complexity of the cache and memory hierarchies is increasing. Therefore, it is becoming more important to analyze the communication behavior of parallel applications when mapping threads to cores, to improve performance and energy efficiency. However, communication is implicit in most programming models for shared memory, which makes it difficult to detect the communication pattern between the threads in an accurate and low-overhead way.**

We propose a new mechanism to detect the communication pattern of shared memory applications by monitoring page table accesses. Combining this mechanism with a dynamic migration algorithm allows mapping to be performed dynamically by the operating system. We implemented our mechanism in the Linux kernel and performed experiments with applications from the NAS Parallel Benchmarks. Results show a reduction of up to 16.7% of the execution time and 63% of the cache misses, compared to the original scheduler of the operating system. Furthermore, we decrease total processor and DRAM energy consumption by up to 14.7% and 28.5%, respectively.

***Keywords*-Mapping; Communication Detection; Page Table; Shared Pages;**

I. Introduction

In recent years, increasing the Thread-Level Parallelism (TLP) has become a key design goal of processor architectures [1]. The rising number of cores per chip and threads per core has brought challenges to the memory and cache subsystems, such as increasing the memory bandwidth and reducing the average memory access latency for the processors. These challenges are being resolved by adding more complex and heterogeneous memory hierarchies to computer architectures. The heterogeneity for memory accesses is expressed in several ways. First, processors include shared and private caches, where access to a local cache is faster than a cache private to a different core. Second, in Non-Uniform Memory Access (NUMA) architectures, memory accesses to the local node are faster than accesses to remote nodes. Furthermore, Non-Uniform Cache Architecture (NUCA) [2] can add another level to the heterogeneity.

In shared memory architectures, communication is performed through memory accesses. The heterogeneity of memory accesses leads to a different communication performance, depending on whether the communication happens through remote or local caches and memories. Since the threads of parallel applications need to communicate to perform their tasks, this heterogeneous communication performance is important to be considered when mapping threads to cores. To take advantage of faster memory accesses, it is necessary to map threads that have a high amount of communication between them to cores that are closer to each other. The main goal of these *Communication-Based Mapping* methods is to improve the communication speed and efficiency, thereby increasing application performance and energy efficiency, which are key factors for current and future hardware architectures [3], [4].

As the communication in shared memory happens implicitly, detecting it in an accurate and efficient way is the main challenge for communication-based mapping methods. Most previous research in this area focuses on static methods using memory traces or binary analysis [5], [6], which present a high overhead. Dynamic methods were also proposed [7], [8], but with several disadvantages, such as low accuracy, the need to modify the source code of the applications or support libraries, or being limited to specific processor architectures.

In this paper, we propose a new method to detect the communication of parallel shared memory applications that works entirely on the operating system level. This method, *Shared Pages Communication Detection (SPCD)*, uses the page table of parallel applications to detect pages that are accessed by more than one thread and mark them as shared. Subsequent accesses to these shared pages are considered as communication among the threads that accessed these pages. In this way, SPCD can detect the communication pattern during the execution of the application, and perform a dynamic migration of threads according to their communication behavior. SPCD has several advantages compared to existing approaches:

Dynamic Detection and Mapping: SPCD detects communication and performs the migration during the execution of the application, without needing prior knowledge about the behavior of the application. Furthermore, it can react to dynamic changes during the execution.

No Modifications to Applications: The mechanism is implemented at the kernel level. No modifications to the source code of the application or support libraries are required. Also, it is independent of the parallelization API.

1530-2075/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPS.2013.57

Hardware-agnostic: The mechanism can easily be adapted to different hardware platforms. It does not depend on architecture-specific hardware counters or features and works with different virtual memory page sizes.

Low Overhead: The overhead on the executed application is low, less than 2%. Furthermore, no expensive operation, such as simulation or memory tracing, is required.

High Accuracy: The communication pattern is measured directly from the memory access behavior of the application, and not estimated using indirect measures, such as cache statistics. Furthermore, the detection is performed on a per-application basis and is not influenced by other applications.

To our knowledge, this is the first communication detection and mapping mechanism for shared memory that combines these properties. We implemented SPCD in the Linux kernel, and performed experiments using the OpenMP implementation of the NAS Parallel Benchmarks (NPB) [9] with 32 threads. Results show substantial gains in performance and energy efficiency, up to 16.7% in execution time and 14.7% in processor energy consumption.

II. Communication in Shared Memory

Figure 1 shows a hardware architecture, consisting of two Intel SandyBridge processors [10]. It has a complex memory hierarchy, formed by three cache levels and a NUMA topology, which causes different communication performance between the threads. In this architecture, there are three possibilities for communication between threads, marked *a*, *b* and *c* in the figure. *a*) Threads that run on the same core can communicate through the fast L1 or L2 caches and have the highest communication performance. *b*) Threads that run on different cores have to communicate through the slower L3 cache, but can still benefit from the fast intra-chip interconnection. *c*) When threads communicate between physical processors or even NUMA nodes, they need to use the slow off-chip interconnection. Hence, the communication

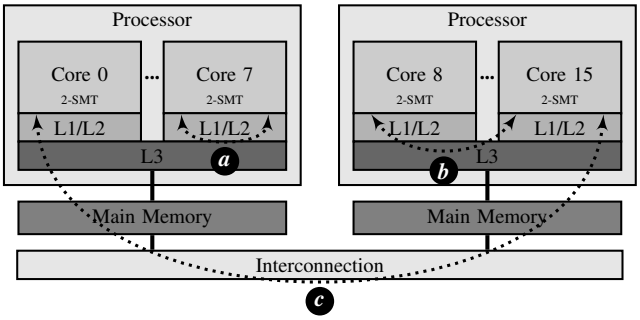


Figure 1: NUMA system architecture with two processors. Each processor consists of eight 2-way SMT cores and has three cache levels. Lines *a*, *b* and *c* show three different communication possibilities between two threads.

performance in this case is the lowest of the three cases possible in this architecture.

A. Communication-Based Mapping

Based on these differences in the communication performance, mechanisms to map threads to cores while taking into account the communication between them can improve performance and energy consumption. The improvements of these *Communication-Based Mapping* mechanisms can be attributed to two factors: reducing cache misses and improving interconnection usage.

The most important effect of Communication-Based Mapping is the reduction of the cache misses, which we classify into three types. The first type of cache miss is the *invalidation miss*, which happens when a cache line used for communication is constantly invalidated through write memory operations of another thread, generating a miss on the next access. By mapping threads that communicate to a shared cache, cache lines used for communication would be considered as private by the cache coherence protocol, thereby not requiring any invalidation messages.

The second type is the *capacity miss*. In shared memory programs, capacity misses often happen when threads that share a cache evict cache lines accessed by other threads [11]. By mapping threads that have a high amount of communication between them to execute on cores that share caches, less cache line evictions are expected, since some cache lines would be used by more than one thread.

The third type of cache miss is the *replication miss*, which happens due to uncontrolled cache line replication. Uncontrolled replication leads to a virtual reduction of the effective size of the caches [12], as multiple caches store the same cache line. By mapping threads that communicate a lot to cores that share a cache, the space wasted with replicated cache lines can be minimized.

The second effect of communication-based 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. Furthermore, the numbers of cache-to-cache transactions and cache line invalidations will be reduced.

The result of these effects is an increase of the performance of an application, as well as a reduction of its energy consumption.

B. Properties of Communication Detection and Mapping Methods

There are several ways to detect communication and perform a mapping. However, some properties are desirable for most applications. The detection mechanism should provide an accurate communication pattern with a low impact on performance. The mechanism should be independent from the implementation of the application, so that it does not

depend on specific libraries or require modifications to the source code. It should also avoid the false communication problem, which can be spatial or temporal. Spatial false communication is similar to 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. Temporal false communication can happen when two threads access the same address, but with a large time difference between the two accesses.

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 complexities rise exponentially, which discourages the use of thread mapping for large groups of threads. Therefore, the communication should be evaluated between pairs of threads, thus generating a *communication matrix*. In this matrix, each cell (i, j) contains the amount of communication between threads i and j. 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.

III. Communication Detection via Shared Pages

In this section, we present our mechanism to detect communication between threads in shared memory architectures, which we call *Shared Pages Communication Detection (SPCD)*. The idea of SPCD is to use the page table of the operating system to detect the communication. The page table is used to translate virtual addresses to physical addresses. To perform this translation, the operating system stores separate translation tables for each process. These tables are indexed by the virtual address, and each entry stores the physical address, as well as memory protection metadata. Threads of parallel applications share the same page table, as they access the same address space. By keeping track of which thread accesses each page, it is possible to use the page table to detect the communication between threads. In the following sections, we first explain the concept of the proposed mechanism, followed by a description of how we implemented it in the Linux kernel.

A. Overview of the Concept

The concept of the SPCD mechanism is to mark virtual pages that have been accessed by more than one thread as shared, and use the number of memory accesses to these shared pages as metric for the communication between threads. To detect shared pages, whenever a page fault is caused by the parallel application, we store the ID of the thread that generated the page fault along with the address of the page. When more than one thread accesses the same page, we consider this page a shared page. Accesses to pages previously marked as shared are considered as communication. We store the communication in a matrix, where each cell of the matrix contains the amount of communication between the threads.

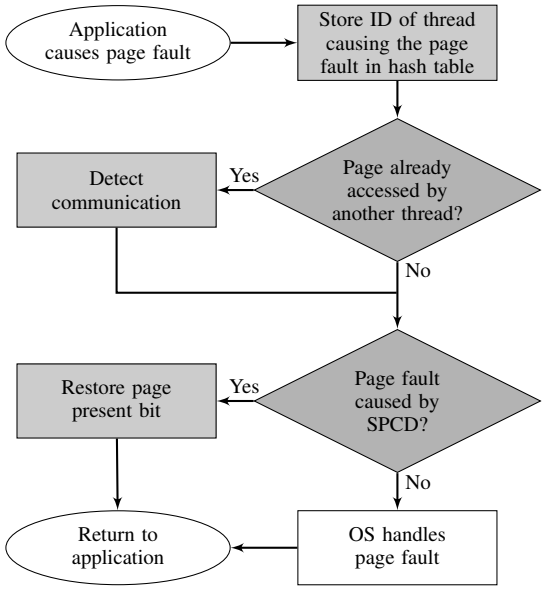


Figure 2: Operation of the SPCD mechanism. The gray boxes were added to the page fault handler of the operating system.

It is important to note that, since all threads share the same page table, there is usually only one page fault per page. To detect the communication pattern, we have to create additional low-overhead page faults, so that more than one fault can happen in the same page. We periodically create these additional page faults using a dynamically adjusting algorithm that keeps the number of additional page faults at a chosen percentage of the total page faults. We create the page faults by clearing the present bit of certain page table entries in the page table, and removing the entry from the Translation Lookaside Buffer (TLB). Afterwards, when a page fault happens on that page, SPCD detects the communication, restores the page present bit, and returns directly to the application. In this way, the overhead from these additional page faults is low. Figure 2 contains an overview of the operation of the mechanism.

To better illustrate the behavior of SPCD with a parallel application, consider an example timeline of the execution shown in Figure 3. After the application starts, thread 0 tries to accesses the virtual address X, which is from a page never accessed before. A page fault is then generated and our mechanism marks that thread 0 accessed the page of addres X. After some time, our mechanism clears the present bit of the page of address X. Thread 1, at a later point, tries to access virtual address X, generating another page fault. SPCD detects that it is responsible for the page fault, restores the present bit of the page and increments the

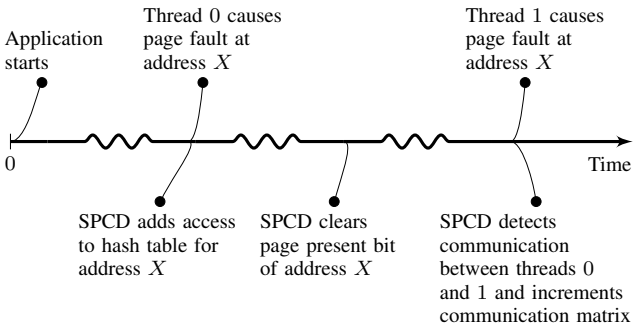


Figure 3: Example timeline of the execution of a parallel application with the SPCD mechanism.

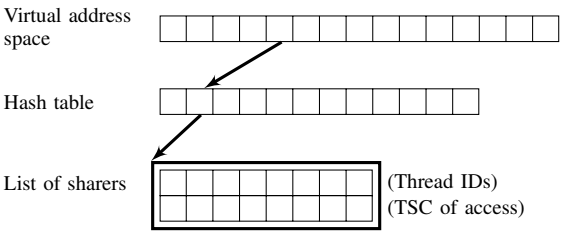


Figure 4: Data structure used to store communication information.

communication matrix in cell $(0, 1)$, which corresponds to threads 0 and 1.

B. Implementation in the Linux Kernel

We implemented the proposed mechanism as a module for the Linux kernel, version 3.2, for the x86-64 architecture. The implementation was divided into two components:

1) Page Fault Handler: We modified the page fault handler to mark the shared pages and detect the communication, as explained in the previous section. The modified handler is only executed in case a parallel application is running, and has no influence on the operation of single-threaded processes. Furthermore, as the communication detection is performed in the virtual memory of a process, there is no interference on the communication pattern from other processes.

For each parallel application, we store which pages are shared and which threads accessed them in a hash table, as illustrated in Figure 4. To be independent from the page size of the hardware architecture, and to reduce the impact of spatial false communication, we chose to store the information outside the page table. In this way, the granularity of the communication detection can be changed. For example, if the page size was 64 MByte, the granularity could be maintained at 4 KByte, because the full address of the faulting memory access is available to the kernel. Each memory region (of the chosen granularity) that the application accesses points to an element of the hash table. In the table, we store the address of the region, the list of sharers and the time stamp of the last access by each thread.

The size of the hash table was chosen to contain $256,000$ elements. With this size, it is possible to cover up to 1 GByte of virtual address space, assuming a granularity of 4 KByte. We use the default hash function of the Linux kernel version 3.2. In case of hash collisions, we overwrite the previous entry to reduce the overhead. The hash table itself consumes 18 MByte of main memory.

2) Additional Page Faults: As mentioned in the overview of the mechanism, we need to create additional page faults to detect communication during the execution. Therefore, we added a kernel thread that periodically creates additional page faults for the parallel application. The thread wakes up at a fixed time interval of 10 ms and performs a page table walk in the page table of the application, clearing the page present bit for a random sample of pages. The thread dynamically adjusts the number of page faults it creates to a chosen ratio of the total page faults.

C. Further Considerations

1) Granularity of the Detection and Spatial False Communication: In our implementation, we used the default page size of the architecture (4 KByte) to determine whether memory was shared. However, as the storing of the shared pages in the hash table is decoupled from the way page faults are managed, it is possible to detect communication at different granularities than the page size. For instance, it would be possible to consider only memory accesses with the same address as communication, and to keep generating additional page faults for the page they reside in. Increasing the granularity reduces the amount of spatial false communication, but a larger hash table is needed to store the same amount of total memory area covered by SPCD.

2) Temporal False Communication: The temporal false communication can be avoided with SPCD in the following way. Whenever a thread accesses the memory region, we save the time stamp of the access. A subsequent access to the same region will only be counted as communication if the access happened within a time window of the original access. If the accesses happen too far apart, they will therefore not be taken into account when detecting the communication.

3) Accuracy of the Mechanism: The accuracy of the detected communication pattern is determined by two factors, the rate at which additional page faults are created and the granularity at which the communication is detected. In our experiments, the values we chose for the page size and additional page fault rate (4 KByte and 10%) were shown to provide both a high accuracy and a low overhead. These values could be modified if an application requires it.

4) Overhead: There are two sources of overhead related to the execution time in the mechanism: creation of additional page faults and the communication detection. The communication detection consists of accesses to the hash table, which has a constant time complexity. The additional page faults, despite being resolved quickly, introduce the need to perform a context switch to the kernel and therefore need to be created at a low rate. In the page fault handler itself, these faults can be resolved by a page table walk, which has a constant time complexity. Creating the additional page fault consists of performing a page table walk as well. In Section V-F, we present an analysis of the runtime overhead caused by the page faults and the communication detection. Regarding memory consumption, the only the overhead is to store the hash table, which has no measurable impact on performance.

5) Architectures with Larger Page Sizes: Currently, most hardware and software architectures use a page size of 4-8 KByte. However, there is a trend to increase the page size, to reduce page faults and TLB misses [13]. We evaluated the mechanism using 4 KByte pages. The implementation of the mechanism does not change for larger pages. In terms of accuracy, this increases the amount of false communication if the granularity for the communication detection is increased to the level of the large page size. However, as mentioned before, the granularity of the communication detection is not tied to the page size, and a lower granularity could be chosen to obtain a higher accuracy.

IV. The Mapping Mechanism

During the execution of the application, we periodically analyze the communication matrix. To dynamically map the threads, we need to provide a way to allow the detection of changes on the communication pattern, as well as to calculate the new mapping.

The calculation of the new mapping is NP-Hard [14]. 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. We developed a thread mapping algorithm based on the graph matching problem. Although we focus on thread mapping in this paper, the mechanisms can be used to perform data mapping as well. Our solution uses the graph matching algorithm from Edmonds [15], which has a polynomial time complexity. To further reduce the overhead of our mapping algorithm, we also developed an algorithm that aims to decrease the amount of times that the mapping algorithm is called. We refer to this algorithm as a *communication filter*. In this section, we will first present the communication filter, followed by the description of the mapping algorithm.

A. Communication Filter

The goal of this filter is to decide if the communication matrix has changed sufficiently to warrant a migration. 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 mapping filter, we limit the size of each subgroup to 2. This reduces the overhead, since we only keep track of one partner thread for each thread.

Every time the 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 $\Theta(N^2)$, where N is the number of threads of the application.

B. The Thread Mapping Algorithm

The 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. This problem can be solved by Edmonds' matching algorithm in polynomial time [15].

To model thread mapping as a matching problem, we use vertices to represent the threads and edges to represent 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 such that the amount of communication is maximized.

If there are only 2 cores sharing a cache, mapping threads to them with the matching algorithm is straightforward. However, there are many architectures in which more than 2 cores share the same cache, or there are more levels of memory hierarchy to be exploited. In these cases, 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. This matrix was generated by the following heuristic function:

$$H_{(x,y),(z,k)} = M_{(x,z)} + M_{(x,k)} + M_{(y,z)} + M_{(y,k)} \quad (1)$$

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. This algorithm does not guarantee that the result will contain the pairs of pairs with the most amount

Table I: Configuration of the machine that hosted the experiments and the SPCD mechanism.

	Parameter	Value
Processors	Processor model	2x Intel Xeon E5-2650, 2.0 GHz
	Number of cores per processor	8, 2-way SMT
	Total number of threads	32
	L1 cache size per core	32 KByte data+32 KByte inst.
	L2 cache size per core	256 KByte
	L3 cache size per processor	20 MByte
Memory	Total amount of memory	32 GByte
	Memory technology	DDR3-SDRAM, 1333MHz
	Page size	4 KByte
SPCD	Granularity	4 KByte
	Additional page faults (approx.)	10%
	Hash table size	256,000 elements

of communication, as the communication matrix does not provide communication information about groups with more than 2 threads. However, it is a reasonable approximation and keeps the time and space complexity polynomial.

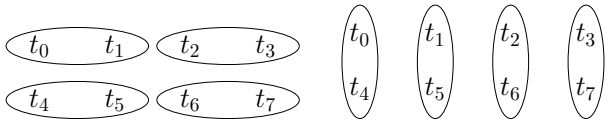
V. Experimental Evaluation

In this section, we first describe the evaluation methodology of SPCD. Afterwards, we show the results of experiments with a producer/consumer benchmark and the NAS Parallel Benchmarks (NPB) [9]. Finally, we analyze the performance overhead imposed by the SPCD mechanism.

A. Methodology of the Experiments

To evaluate the mechanism, we first verify the dynamic detection using a producer/consumer benchmark that will be introduced in the next subsection. For the rest of the evaluation, we use the OpenMP implementation of the NAS Parallel Benchmarks (NPB) [9], version 3.3.1, with the A input size. All benchmarks are executed using 32 threads. We present three results for the benchmarks: the communication patterns, performance statistics (execution time, L2/L3 MPKI and cache-to-cache transactions) and statistics about energy consumption (processor and DRAM energy). We execute each experiment 10 times and show the average values, as well as the confidence interval for a confidence level of 95% in a Student's t-distribution.

The machine used to run the benchmarks is a dual-node NUMA machine. The nodes consist of an Intel Xeon E5-2650 processor [10], with eight 2-way SMT cores. The L1 and L2 caches are private to each core, whereas the L3 cache is shared among all cores of the processor. The system architecture is depicted in Figure 1. We implemented SPCD as a module for the Linux kernel, version 3.2. The machine and SPCD configurations are summarized in Table I.



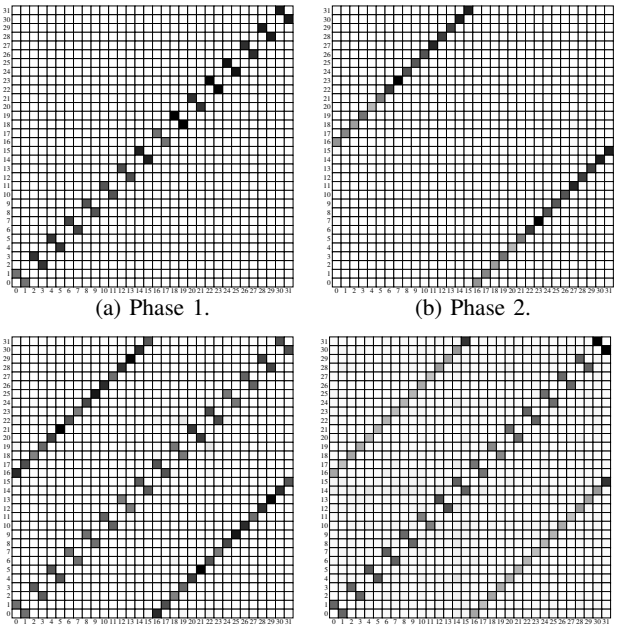
(a) Phase 1. (b) Phase 2.

Figure 5: The two phases of the producer/consumer benchmark. Circled threads communicate with each other.

B. Verifying SPCD with a Producer/Consumer Benchmark

To verify the operation of SPCD and the detection of dynamic application behavior, we created a producer/consumer benchmark. It consists of pairs of threads, where each pair communicates with each other through a shared vector. To create a dynamic behavior, the communication between the threads changes periodically between two phases. These phases are depicted in Figure 5. In the first phase, neighboring threads are communicating with each other. In the second phase, more distant threads are communicating. With this behavior, the best mapping changes for each phase.

We executed this benchmark with 16 pairs of producer/consumer threads on the machine described previously. The communication matrices detected by SPCD are shown in Figure 6. On the x and y-axes, the thread IDs are



(a) Phase 1. (b) Phase 2.
(c) Transition between phases. (d) Overall Communication.

Figure 6: Communication matrices of the producer/consumer benchmark. The x and y-axes contain the thread IDs, while the cells show the amount of communication between two threads. Darker cells indicate a higher amount of communication.

displayed. The matrix cells show the amount of communication between two threads, where a darker shade indicates a higher amount of communication.

In the first phase, SPCD correctly detects that the neighboring threads are communicating with each other, as can be seen in Figure 6a. Figure 6b shows the communication matrix for the second phase, where the distant threads are communicating. When the phases change, there is a transitional period in which there are traces of both phases visible (Figure 6c). The overall communication pattern, without considering the dynamic behavior, can be seen in Figure 6d. This is the communication pattern that a static detection mechanism would generate, it does not allow to create an improved mapping for this dynamic application. The results from this experiment show that SPCD is able to detect the dynamic behavior of an application.

C. NAS Communication Patterns

The communication patterns of the NAS benchmarks detected by SPCD are shown in Figure 7. We show only the final communication matrices, as the communication patterns do not change after a short period of initialization. The communication patterns of the applications can be classified as *homogeneous* or *heterogeneous*. Homogeneous communication means that the threads present a similar amount of communication among each other. In applications with heterogeneous communication patterns, there are threads that communicate more with a subgroup of threads.

In BT, MG, LU, SP and UA, most of the communication happens between neighboring threads, and their communication patterns are heterogeneous. This is common in applications that are based on domain decomposition, where most of the communication happens between neighboring threads and most of the shared data is located on the borders of each sub-domain. Although the amount of communication is not very high, CG and DC also present a slightly heterogeneous pattern for neighboring threads.

FT and IS show a homogeneous communication behavior, with no clear pattern visible. EP has a homogeneous communication pattern as well. Furthermore, the total amount of communication is very low, with several threads not communicating at all. This is a common behavior in applications that focus on the raw computational performance.

For the performance evaluation, we expect the benchmarks with a clear, heterogeneous pattern to benefit more from communication-based mapping, since threads that communicate would be closer to each other in the memory hierarchy. On the other hand, in benchmarks with homogeneous patterns, there are no significant differences in the amount of communication between the threads, such that there is no mapping that optimizes the communication.

D. NAS Performance Results

To analyze the performance improvements of the SPCD mechanism, we show the execution time and L2/L3 cache

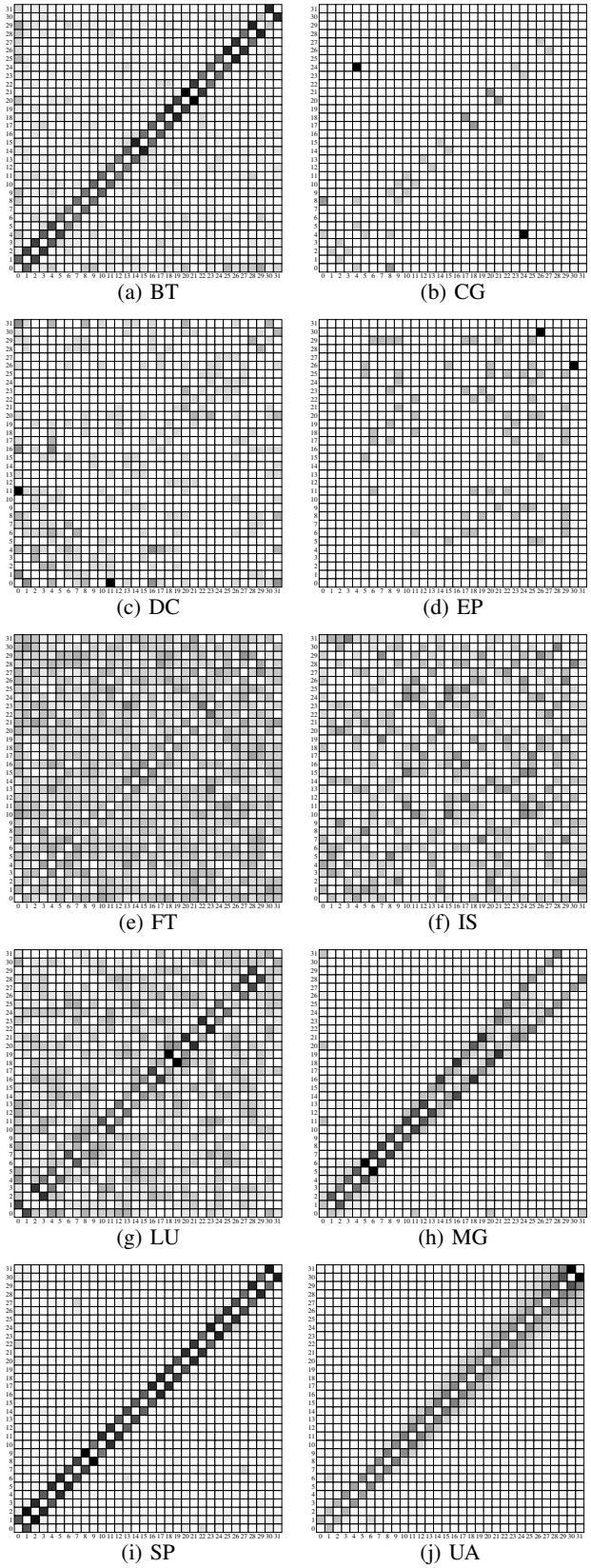


Figure 7: Communication matrices of the NAS benchmarks.

MPKI obtained by hardware counters evaluated with the PAPI framework [16]. To measure the traffic on the interconnections, we evaluate the number of cache-to-cache transactions with the help of Intel VTune [17]. We show the results of SPCD, the original Linux scheduler, a random mapping and an oracle mapping.

Operating system mapping: This mapping uses the original scheduler of the Linux kernel. It represents the baseline for our experiments.

Random mapping: We generated 10 different mappings randomly, one for each execution, and execute all benchmarks with the same 10 random mappings. This mapping presents an evaluation without the overhead of migrations introduced by the operating system.

Oracle mapping: 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.

Figures 8, 9, 10 and 11 present the execution time, L2 and L3 cache MPKI and cache-to-cache transactions, respectively. The graphics are normalized to the result of the original scheduler of the Linux operating system.

BT, LU, SP and UA are applications that we classified as having heterogeneous communication patterns. In these applications, our mechanism performs better than the operating system and the random mapping. Additionally, the results of SPCD for these benchmarks are very close to the oracle scheduler. It is important to note that the numbers of cache misses and cache-to-cache transactions were greatly reduced as well, at a higher rate than the execution time. For the SP benchmark, SPCD achieved the highest improvements, reducing the execution time by 16.7%. In general, L2 and L3 misses were reduced by up to 18% and 63%, while cache-to-cache transactions were reduced by up to 76%.

CG and DC show performance benefits as well, though in a smaller amount, even slightly outperforming the oracle mapping. This can happen because the oracle mapping only optimizes the mapping in terms of communication between threads, but does not take into other factors, such as contention on the execution units and interference from the operating system, among others. In MG, despite having a heterogeneous pattern, SPCD does not gain performance and has a longer execution time than the random mapping.

For IS, EP, and FT, no performance gains are expected due to their homogeneous communication patterns, and SPCD reduces their performance slightly. EP and FT show an increase in the number of cache-to-cache transactions. However, the absolute numbers are very low (Table II).

In summary, we achieved good results for the applications that have a heterogeneous communication behavior, improving performance for seven out of ten benchmarks.

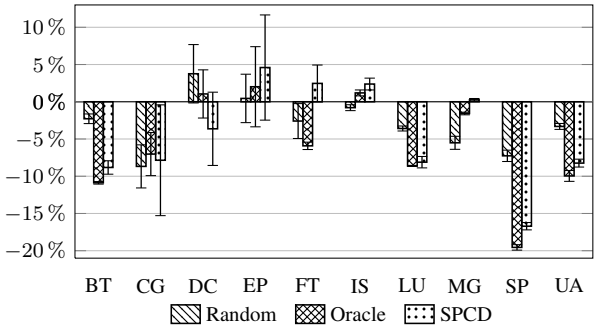


Figure 8: Execution time (normalized to the OS).

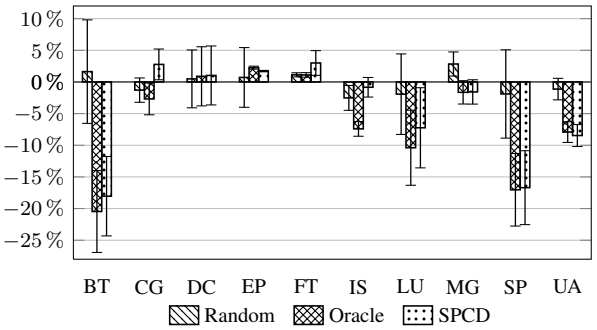


Figure 9: L2 cache MPKI (normalized to the OS).

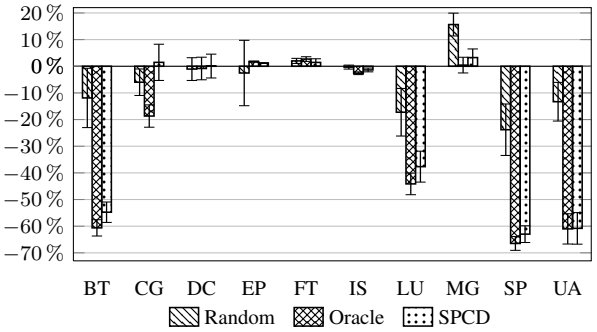


Figure 10: L3 cache MPKI (normalized to the OS).

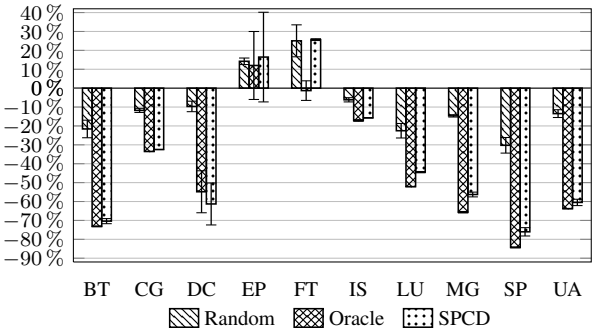


Figure 11: Cache-to-cache transactions (normalized to the OS).

E. NAS Energy Results

Another important factor for the evaluation is the energy savings achievable by the mechanism. A pure reduction of the execution time will lead to a reduction of the energy consumption with the same ratio, in most cases. However, communication-based mapping also causes a more efficient use of the on-chip and off-chip interconnections, which can lead to more energy savings.

To evaluate the energy consumption, we use the Running Average Power Limit (RAPL) hardware counters introduced by Intel in the SandyBridge processor architecture [10] and compare the results of SPCD to the three mappings described in the previous section. We show the results for total processor and DRAM energy consumed during the execution of each benchmark (Figures 12 and 13), as well as the processor and DRAM energy per instruction (Figures 14 and 15).

For all benchmarks except IS, we reduce total processor energy consumption (Figure 12). Similar to the performance results, we save more energy for benchmarks with a heterogeneous pattern. For SP, we again achieve the best result, with a reduction of 14.7%. In DRAM energy results (Figures 13), we see a very similar behavior. All benchmarks except CG save DRAM energy, with UA achieving a reduction of 28.5%. In most cases, our mechanism achieves a similar result as the oracle, even outperforming it in some benchmarks.

The energy per instruction results shown in Figures 14 and 15 correlate with the total energy consumption. Energy was saved in almost all cases, with results close to the oracle mapping. These energy per instruction results also show that energy consumption was improved not only by reducing the execution time, but also making the execution more efficient, which is important for current and future architectures [18].

F. Overhead of SPCD

As the SPCD mechanism creates additional page faults and needs to keep track of communication information during the runtime, it causes an overhead for the application. An additional source of overhead is the calculation of the mapping and the migration of the threads. These two sources of overhead were evaluated by measuring the time spent in the SPCD and mapping algorithm and comparing this time to the total execution time. The amount of the overhead is shown in Figure 16, with absolute results shown in Table II.

For all benchmarks, the overhead imposed by the SPCD mechanism is less than 1.5%. As the mapping algorithm is executed only when a communication pattern changes, it has a very low overhead of less than 0.5% for all benchmarks. The total overhead of SPCD and the mapping mechanism is therefore less than 2% for all benchmarks.

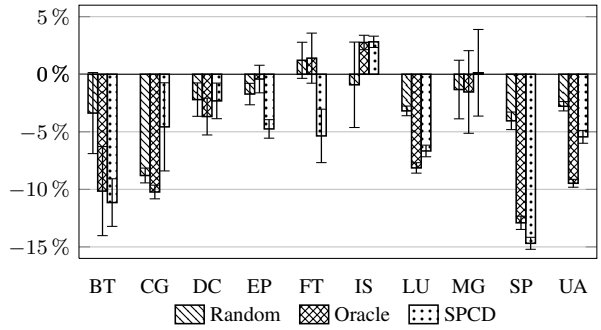


Figure 12: Total processor energy (normalized to the OS).

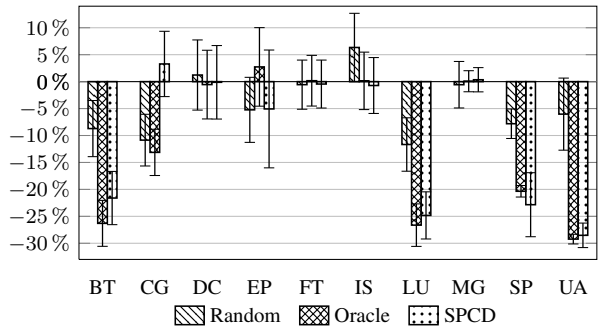


Figure 13: Total DRAM energy (normalized to the OS).

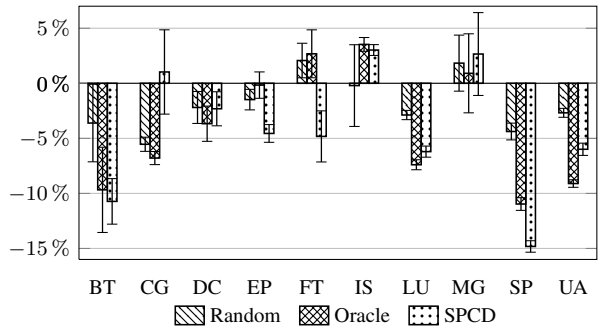


Figure 14: Processor energy per instruction (normalized to the OS).

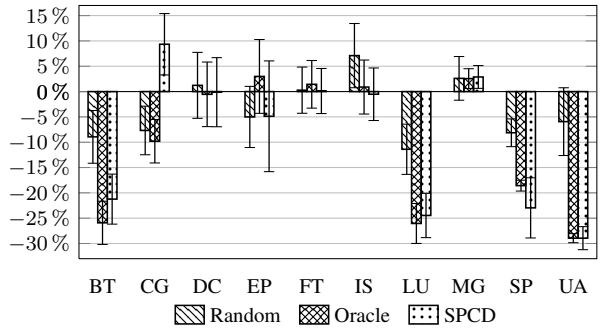


Figure 15: DRAM energy per instruction (normalized to the OS).

Table II: Absolute results achieved by the SPCD mechanism. For the performance and energy results, the difference to the operating system mapping is given in parentheses.

	Parameter	BT	CG	DC	EP	FT	IS	LU	MG	SP	UA
	Communication pattern	Hetero-geneous	Hetero-geneous	Hetero-geneous	Homo-geneous	Homo-geneous	Homo-geneous	Hetero-geneous	Hetero-geneous	Hetero-geneous	Hetero-geneous
Performance	Execution time (s)	5.35 (-8.8%)	0.22 (-7.8%)	104.37 (-3.6%)	0.93 (+4.6%)	0.83 (+2.4%)	0.26 (+2.6%)	3.84 (-8.1%)	1.15 (+0.3%)	5.91 (-16.7%)	4.16 (-8.2%)
	L2 cache MPKI	2.44 (-18.0%)	16.27 (+2.8%)	17.39 (+1.0%)	0.16 (+1.7%)	16.82 (+3.0%)	4.86 (-0.8%)	3.60 (-7.2%)	9.48 (-1.6%)	9.42 (-16.7%)	4.03 (-8.5%)
	L3 cache MPKI	0.20 (-54.8%)	0.24 (+1.5%)	9.46 (+0.1%)	0.02 (+1.2%)	0.93 (+1.4%)	2.36 (-1.4%)	0.52 (-37.7%)	2.13 (+3.3%)	0.58 (-63.0%)	0.28 (-60.8%)
	Cache-to-cache transactions (k)	42500 (-70.4%)	2002 (-32.5%)	6000 (-61.3%)	131 (+16.4%)	918 (+25.7%)	1695 (-15.7%)	177500 (-44.4%)	1763 (-56.3%)	72500 (-76.0%)	132000 (-60.5%)
Energy	Total processor energy (J)	347.03 (-11.1%)	9.65 (-4.6%)	3683.14 (-2.3%)	49.44 (-4.7%)	45.80 (-5.4%)	9.20 (+2.8%)	266.07 (-6.7%)	53.20 (+0.1%)	395.80 (-14.7%)	284.87 (-5.4%)
	Total DRAM energy (J)	13.33 (-21.6%)	0.41 (+3.5%)	320.36 (-0.1%)	1.67 (-5.1%)	3.54 (-0.4%)	0.69 (-0.9%)	12.39 (-24.8%)	5.44 (+0.4%)	23.23 (-22.8%)	9.15 (-28.5%)
	Proc. energy per inst. (nJ)	1.83 (-10.7%)	2.21 (+1.0%)	8.66 (-2.3%)	2.14 (-4.6%)	3.37 (-4.8%)	3.77 (+3.0%)	2.36 (-6.2%)	7.41 (+2.7%)	3.43 (-14.8%)	2.46 (-6.0%)
	DRAM energy per inst. (nJ)	0.07 (-21.2%)	0.09 (+9.6%)	0.75 (-0.1%)	0.07 (-4.9%)	0.26 (+0.1%)	0.28 (-0.7%)	0.11 (-24.5%)	0.76 (+2.9%)	0.20 (-23.0%)	0.08 (-28.9%)
SPCD	Number of migrations	6	0	1	1	1	0	6	1	4	1
	Detection Overhead	0.41%	1.04%	1.31%	1.22%	1.46%	1.34%	0.73%	1.14%	1.05%	1.29%
	Mapping Overhead	0.16%	0.42%	0.01%	0.14%	0.28%	0.08%	0.20%	0.11%	0.13%	0.03%

VI. Related Work

In this section, we contextualize the state of art in communication-based mapping mechanisms in shared memory and compare them to our proposal.

A. Static Thread Mapping

In [19], a technique to collect the communication pattern of the threads of parallel applications based on shared memory is introduced. 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. In [20], the authors also analyze the communication pattern of applications, but using the Pin dynamic binary analysis (DBA) tool [5]. The authors of these two papers do not evaluate the performance, they analyze only the communication behavior.

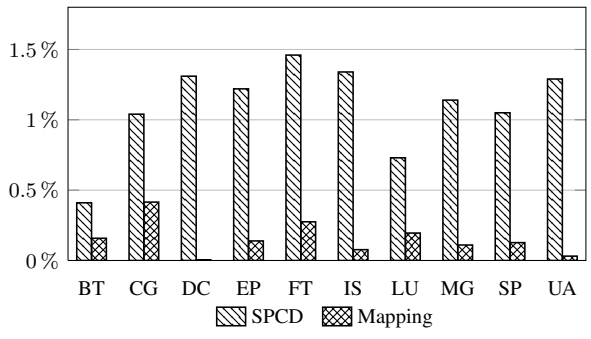


Figure 16: Overhead of SPCD and the mapping mechanism, as a percentage of the total execution time.

In [6], 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 and measured the performance improvement. These methods allow accurate detection of the communication patterns, as they have access to information about all memory accesses of a given application. Despite improving performance, they are infeasible for real applications, as they require simulation or DBA, which demand a lot of processing time and computational resources.

B. Dynamic Thread Mapping

In [7], the authors show that hardware performance counters already present in current processors may be used to dynamically map parallel applications. They schedule threads taking into account an indirect estimate of 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. To decrease the overhead of the proposed mechanism, the mapping system is only triggered after the number of core stall cycles and cache misses 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%. However, memory accesses resolved by local cache memories or the main memory are not considered when detecting the communication, generating an incomplete communication pattern.

The ForestGOMP library [8] integrates into the OpenMP 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 NUMA machines, as in these 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. They also evaluated the performance with the *Stream* benchmark [21], which measures the memory bandwidth, improving execution time and bandwidth by up to 80%. The hardware counters the authors used to guide the thread and data mapping only indirectly estimate the communication patterns. Also, their work is limited to applications that are based on OpenMP.

In [22], the Translation Lookaside Buffer (TLB) was used to dynamically detect the communication pattern of parallel applications. The TLB 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 was detected by searching all TLBs for matching entries. Although this method provides a high accuracy, many current processor architectures, such as x86 and ARM, require hardware modifications to be able to use it.

VII. Conclusions

In this paper, we introduced SPCD, a mechanism to detect the communication between threads of parallel applications in shared memory environments. We used the detected communication pattern to perform a mapping of threads to cores. Compared to previous work, this mechanism is able to dynamically detect the communication with a low overhead and high accuracy, while needing no hardware support or modifications to the parallel applications. Our mechanism performs the communication detection by making use of the virtual memory subsystem of the Linux kernel.

In experiments with the NAS parallel benchmarks, SPCD was able to identify the communication patterns of all applications and migrate them during the execution. As expected, applications that communicated more and presented heterogeneous communication patterns showed the largest improvements. Results show improvements of the execution time of up to 16.7%, reducing cache misses by up to 63%. Processor and DRAM energy consumption were improved by up to 14.7% and 28.5%, respectively.

Acknowledgment

We thank Marco A. Z. Alves, Francis B. Moreira, and the anonymous reviewers for their comments and suggestions. This work was partially supported by CNPq, FAPERGS and CAPES.

References

[1] K. Asanovic, B. C. Catanzaro, D. A. Patterson, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[2] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5, p. 211, Dec. 2002.

[3] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, pp. 67–77, May 2011.

[4] P. W. Coteus, J. U. Knickerbocker, C. H. Lam, and Y. A. Vlasov, "Technologies for exascale systems," *IBM Journal of Research and Development*, vol. 55, no. 5, pp. 14:1–14:12, September-October 2011.

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

[6] M. Diener, F. L. Madruga, E. R. Rodrigues, M. A. Z. Alves, J. Schneider, P. O. A. Navaux, and H.-U. Heiß, "Evaluating thread placement based on memory access patterns for multi-core processors," in *IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2010, pp. 491–496.

[7] R. Azimi, D. K. Tam, L. Soares, and M. Stumm, "Enhancing operating system support for multicore processors by using hardware performance monitoring," *SIGOPS Operating Systems Review*, vol. 43, pp. 56–65, April 2009.

[8] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst, "Structuring the execution of openmp applications for multicore architectures," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010, pp. 1–10.

[9] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS Parallel Benchmarks and Its Performance," Technical Report NAS-99-011, NASA Ames Research Center, Tech. Rep. October, 1999.

[10] Intel, "2nd Generation Intel ® Core ™ Processor Family," Tech. Rep. September, 2012. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/2nd-gen-core-desktop-vol-1-datasheet.pdf

[11] X. Zhou, W. Chen, and W. Zheng, "Cache sharing management for performance fairness in chip multiprocessors," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2009, pp. 384–393.

[12] Z. Chishti, M. Powell, and T. Vijaykumar, "Optimizing replication, communication, and capacity allocation in cmps," in *International Symposium on Computer Architecture (ISCA)*, June 2005, pp. 357–368.

[13] A. Morari, R. Gioiosa, R. W. Wisniewski, B. S. Rosenburg, T. a. Inglett, and M. Valero, "Evaluating the Impact of TLB Misses on Future HPC Systems," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2012, pp. 1010–1021.

[14] S. Bokhari, "On the mapping problem," *IEEE Transactions on Computers*, vol. C-30, no. 3, pp. 207–214, March 1981.

[15] C. Osiakwan and S. Akl, "The maximum weight perfect matching problem for complete weighted graphs is in pc," in *IEEE Symposium on Parallel and Distributed Processing (SPDP)*, December 1990, pp. 880–887.

[16] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A Portable Programming Interface for Performance Evaluation on Modern Processors," *International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, 2000.

[17] Intel, "Intel® VTune Amplifier XE 2013," 2013. [Online]. Available: http://software.intel.com/en-us/intel-vtune-amplifier-xe

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

[19] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterisation of splash-2 and parsec," in *IEEE International Symposium on Workload Characterization (IISWC)*, October 2009, pp. 86–97.

[20] C. Bienia, S. Kumar, and K. Li, "Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *IEEE International Symposium on Workload Characterization (IISWC)*, September 2008, pp. 47–56.

[21] Stream, "Stream," http://www.cs.virginia.edu/stream/, October 2011.

[22] E. H. M. Cruz, M. Diener, and P. O. A. Navaux, "Using the translation lookaside buffer to map threads in parallel applications based on shared memory," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2012, pp. 532 –543.