UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

EDUARDO HENRIQUE MOLINA DA CRUZ

Dynamic Detection of the Communication Pattern in Shared Memory Environments for Thread Mapping

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Prof. Dr. Philippe O. A. Navaux
Advisor

Porto Alegre, Março 2012

CIP – CATALOGING-IN-PUBLICATION

Eduardo Henrique Molina da Cruz,

Dynamic Detection of the Communication Pattern in Shared Memory Environments for Thread Mapping /

Eduardo Henrique Molina da Cruz. – Porto Alegre: PPGC da UFRGS, 2012.

86 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2012. Advisor: Philippe O. A. Navaux.

1. Thread mapping. 2. Parallel computer architectures. 3. Shared memory. 4. Communication. 5. Cache memory. 6. Cache coherence protocols. 7. TLB. I. Navaux, Philippe O. A.. II. Título.

"I believe in intuition and inspiration.
Imagination is more important than knowledge.
For knowledge is limited, whereas imagination embraces the entire world,
stimulating progress, giving birth to evolution.
It is, strictly speaking, a real factor in scientific research."

— ALBERT EINSTEIN

AGRADECIMENTOS

Agradeço minha mãe por seu esforço e pelo apoio nessa jornada da Pós-Graduação, sendo que dedico este trabalho a ela. Também agradeço a meu pai, a quem pude aumentar o grau de contato agora que moramos mais próximos. Não esqueço também da importância do meu primo (que está mais para irmão) Gustavo, da minha tia Lourdes, meu tio Luiz, e de todos meus outros familiares. Relembro aqui também de meus grandes amigos Guilherme, Ramon, Christian e Igor, que, apesar da distância e falta de contato, são pessoas muito importantes em minha vida.

A meus amigos do GPPD, obrigado pelas dicas, discussões e, não menos importantes, momentos de descontração durante estes 2 anos de mestrado. Um agradecimento especial ao professor Navaux pela orientação e oportunidade que me ofereceu. Ressalto também os amigos que tiveram um contato mais próximo com este trabalho. O Marco pelos conselhos e puxões de orelha, o Matthias pela ajuda no desenvolvimento de algumas das propostas, e o Kassick pelo auxílio na correção.

Aos familiares e amigos que aqui não citei, obrigado por tudo.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

SMP	Symmetric Multi-Processor
SMT	Simultaneous Multithreading
NUMA	Non-Uniform Memory Access
SIMD	Single Instruction Multiple Data
SPMD	Single Program Multiple Data
ISA	Instruction Set Architecture
I/O	Input/Output
MIPS	Million Instructions per Second
GPU	Graphic Processing Unit
NoC	Network-on-Chip
DBA	Dynamic Binary Analysis

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The threads of parallel applications cooperate in order to fulfill their tasks, thereby communication is performed among themselves. The communication latency between the cores in a multiprocessor architecture differs depending on the memory hierarchy and the interconnections. With the increase in 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, which makes 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 master thesis, we introduce two novel light-weight mechanisms to find the communication pattern of threads. The first mechanism makes use of the information about shared cache lines provided by cache coherence protocols. The second mechanism makes use of the Translation Lookaside Buffer (TLB) to detect which memory pages each core is accessing. Both our mechanisms rely 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 mechanisms with the NAS Parallel Benchmarks (NPB) and obtained accurate representations of the communication patterns. We generated thread mappings from the detected communication patterns using a mapping algorithm. Mapping is a NP-Hard problem. Therefore, in order to achieve a polynomial complexity, we designed 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% compared to the original scheduler of the operating system. The number of cache misses, cache line invalidations and snoop transactions were reduced by up to 31.9%, 41% and 65.4%, respectively.

Keywords: Thread mapping, parallel computer architectures, shared memory, communication, cache memory, cache coherence protocols, TLB.

Detecção Dinâmica do Padrão de Comunicação em Ambientes de Memória Compartilhada para o Mapeamento de Threads

RESUMO

As *threads* de aplicações paralelas cooperam a fim de cumprir suas tarefas, dessa forma, comunicação é realizada entre elas. A latência de comunicação entre os núcleos em arquiteturas multiprocessadas diferem dependendo da hierarquia de memória e das interconexões. Com o aumento do número de núcleos por *chip* e número de *threads* por núcleo, esta diferença entre as latências de comunicação está aumentando. Portanto, é importante mapear as *threads* de aplicações paralelas levando em conta a comunicação entre elas.

Em aplicações paralelas baseadas no paradigma de memória compartilhada, a comunicação é implícita e ocorre através de acessos à variáveis compartilhadas, o que torna difícil a descoberta do padrão de comunicação entre as *threads*. Mecanismos tradicionais usam simulação para monitorar os acessos à memória realizados pela aplicação, requerendo modificações no código fonte e aumentando drasticamente a sobrecarga.

Nesta dissertação de mestrado, são introduzidos dois mecanismos inovadores com uma baixa sobrecarga para se detectar o padrão de comunicação entre *threads*. O primeiro mecanismo faz uso de informações sobre linhas compartilhadas de caches providas por protocolos de coerência de cache. O segundo mecanismo utiliza a *Translation Lookaside Buffer* (TLB) para detectar quais páginas de memória cada núcleo está acessando. Ambos os mecanismos dependem totalmente do *hardware*, o que torna o mapeamento de *threads* transparente aos programadores e permite que ele seja realizado dinamicamente pelo sistema operacional. Além disto, nenhuma tarefa de alta sobrecarga, como simulação, é requerida.

As propostas foram avaliadas com o *NAS Parallel Benchmarks* (NPB), obtendo representações precisas dos padrões de comunicação. Mapeamentos para as *threads* foram gerados utilizando os padrões de comunicação descobertos e um algoritmo de mapeamento. O problema do mapeamento é NP-Difícil. Portanto, de forma a se atingir uma complexidade polinomial, o algoritmo empregado é heurístico, baseado no algoritmo de emparelhamento de grafos de Edmonds. Executando as aplicações com o mapeamento resultou em um ganho de desempenho de até $15,3\%$. O número de faltas na cache, invalidações em linhas de cache e transações de espionagem foram reduzidos em até $31,9\%$, 41% e $65,4\%$, respectivamente.

Palavras-chave: mapeamento de threads, arquiteturas paralelas de computadores, memória compartilhada, comunicação, memória cache, protocolos de coerência de cache, TLB.

1 INTRODUCTION

The performance of sequential computing is reaching its limits. The rate of increase of instruction level parallelism in superscalar architectures is reducing each year, and the number of pipeline stages became so high that it is difficult to break the execution in more steps to increase the operation frequency (SHALF; DOSANJH; MORRISON, 2011). Additionally, in the past few years, the power consumption of the processors started to play a big role on the design of new high performance architectures. Aggressive instruction speculation, out-of-order execution, as well as deep pipelines, require high amounts of energy. Therefore, high performance architectures focus on thread level parallelism, and are based on several processor cores executing in parallel. This makes the number of cores more relevant than the individual performance of each core.

As the industry relies on parallelism, the increase of the number of cores in multicore architectures is one of the adopted solutions. This makes the memory wall problem (HENNESSY; PATTERSON, 2007) more relevant, since more bandwidth between the cores and the main memory is required. Currently, memory hierarchies with several levels of cache memories are employed to overcome this issue. However, with the upcoming increase of the number of cores, it is expected an aggravation of the memory wall problem. Therefore, novel solutions are required to allow the performance to scale (COTEUS et al., 2011; TORRELLAS, 2009).

One of the main concerns regarding multicore architectures is the communication between threads (ZHAI et al., 2011). Communication implies in data movement among the cores, leading to performance loss and energy consumption (BORKAR; CHIEN, 2011). Therefore, it is important to research and develop mechanisms to optimize the communication. Some studies focus on hardware, suggesting new interconnections and network topologies (STEVENSON; CONN, 2011; AJIMA; SUMIMOTO; SHIMIZU, 2009), while others focus on software, to improve data locality (RIBEIRO et al., 2010). As the scope of the problem is wide, there are several ways to deal with it.

1.1 Scope of this research

In multicores architectures, some levels of the memory hierarchy are shared by more than one core, which causes the communication latency among the cores to be different. Additionally, in some architectures, there is more than one processor, in which each processor has several cores. This increases the number of levels of the memory hierarchy, since cores of the same processor communicate faster than cores of different processors. Future interconnections introduced by *Network-on-Chip* (NoC) (DE MICHELI; BENINI, 2006; FREITAS et al., 2007) are expected to increase the difference in the communication latencies among the cores. Several levels of memory hierarchy also impose a high overhead on cache coherence protocols (CHISHTI; POWELL; VIJAYKUMAR, 2005), which are responsible to keep the data integrity among all the cache memories.

Thread mapping helps to improve performance by mapping the threads on cores according to some policy, such that the usage of the resources is optimized. By mapping the threads considering the amount of communication between them, the communication latency is reduced, since threads that communicate are mapped to nearby cores on the memory hierarchy. The overhead of cache coherence protocols is also reduced, because the number of cache-to-cache and invalidation transactions is thereby decreased (ALVES; FREITAS; NAVAUX, 2009).

The level of difficulty to map the threads depends on the parallel programming paradigm. When the paradigm is messaging passing (RODRIGUES et al., 2009), detecting the communication pattern is rather straightforward, and is accomplished by monitoring the origin and destination fields for each message. However, when the paradigm is shared memory, the communication between the threads is implicit and it happens every time a thread reads or writes data that has been previously accessed by other threads. Therefore, mapping shared memory based applications is much more challenging. Another factor that influences on the difficulty is if the mapping is made statically or dynamically. In static thread mapping, the information on the communication pattern is gathered by profiling the application in previous executions using controlled environments such as simulators. In dynamic thread mapping, the information on the communication pattern must be gathered while running the application, a much more difficult task.

It is important to take into account the viability of the proposed methods. For instance, architectures such as of the graphic processing units (GPU) provide high performance, but requires complex software (LIU; ZHANG; SHEN, 2009). Some mechanisms present high overhead steps of profiling, discouraging their usage (WANG; O'BOYLE, 2009). Other mechanisms require modifications on the source code of the applications, increasing the complexity of the programming (IBRAHIM, 2010). Such mechanisms also reduces the portability of the applications, since the modifications usually depend on the target architecture. Furthermore, relying on programmers is not desirable, because inexperienced programmers may insert wrong annotations.

1.2 Proposal

In this master thesis, we propose two different mechanisms to dynamically detect the communication pattern among the threads of shared memory based applications. Both mechanisms allow the thread mapping to be performed dynamically by the operating system and do not require simulation or any changes to the source code of the applications. Our first proposed mechanism makes use of cache coherence protocols. It is based on two fundamental ideas. First, a cache line that is shared by more than one cache indicates that more than one core is accessing the same memory location, which represents a communication. Second, cache coherence protocols keep track of shared cache lines. By adding some minimal extra hardware to the original cache coherence system, we allow the hardware to count the number of memory accesses to shared cache lines.

Our second proposed mechanism to find the communication pattern 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 per core, the communication pattern could be detected by searching all TLBs for matching entries. We developed mechanisms for both software-managed and hardware-managed TLBs, covering most of the current architectures.

1.3 Organization of the text

The text is organized as follows. Chapter 2 explains how thread mapping works and presents some related work. Chapter 3 shows our proposed mechanisms for dynamic detection of the communication pattern. Chapter 4 shows the methodology we adopted to evaluate our proposals. Chapter 5 contains the results of the experiments. Finally, Chapter 6 draws our conclusions and future work.

2 THREAD MAPPING

The basic goal of thread mapping is to optimize the usage of the available resources. When dealing with threads, the parallel programming paradigm used is shared memory, in which all the communication among the threads is performed by accesses to the memory. Therefore, memory is the main resource to be considered when mapping the threads of applications based on this paradigm. As multi-core architectures have memory hierarchies with several levels of cache memories, some cache lines may be present in more than one cache. These cache lines are said to be replicated, or shared. Keeping data integrity among all the caches is responsibility of cache coherence protocols (STALLINGS, 2006). To do this, cache coherence protocols store information about the state of each cache line, such as if the line is shared or modified. They also send messages through the interconnections to the caches when some data is requested or modified.

In snoop protocols (EGGERS; KATZ, 1989), the information about the state is kept by the caches and messages are exchanged between them. In directory protocols (AGARWAL et al., 1988), the directory is responsible to keep this information and to send the messages to the caches. However, in both snoop and directory protocols, the overhead generated by the messages sent through the interconnections is high. For instance, 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. If the cache coherence protocol is based on invalidation, such as MESI (STALLINGS, 2006), and the reader and writer do not share a cache, an invalidation message would be sent to the reader every time the writer writes the data. As a result, the reader would always receive a cache miss when reading, thereby requiring more coherence traffic on the interconnections, since the cache of the reader would have to retrieve the data from the cache of the writer on every access.

One way to reduce this overhead is by mapping the threads that communicate on cores that are close to each other in the memory hierarchy. In the previous example, no coherence traffic would be generated if the writer and reader shared a cache. It is important to note that write operations impact more on performance than read operations, as all writes to shared cache lines invalidate the corresponding lines on the other caches. Furthermore, 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.

As mentioned before, thread mapping reduces the cache misses generated by the invalidation messages. This type of cache miss is called invalidation miss. Other side effects are the reduction of capacity misses and replication misses. Capacity misses are cache misses that happen when data is accessed for the first time and was not already fetched from memory. In a shared cache, threads compete for cache lines and evict cache lines from each other (ZHOU; CHEN; ZHENG, 2009). By mapping threads that communicate to shared caches, this competition is reduced. Replication misses are cache misses that happen due to uncontrolled cache line replication. As stated in CHISHTI; POWELL; VIJAYKUMAR (2005), uncontrolled replication leads to a virtual reduction of the effective size of the caches, as some of their space would be used to store the same cache lines. By mapping threads that communicate to cores that share a cache, the space wasted with replicated cache lines is minimized, leading to a reduction of the cache misses.

The mapping can also be applied to processes. Processes, contrary to threads, do not share memory among themselves by default. In order to communicate, the processes send and receive messages to each other. This parallel programming paradigm is called message passing (SEBESTA, 2009). The discovery of the communication pattern of message passing based applications is straightforward compared to shared memory based applications. This happens because the messages keep fields that explicitly identify the source and destination. We discuss some related work about message passing in Sections 2.2.4 and 2.3.3.

Regarding architectures with non-uniform memory access characteristics (NUMA), besides thread mapping, data mapping is also important (TERBOVEN et al., 2008; RIBEIRO et al., 2009). The data mapping is required in NUMA because the access latencies to the memory banks are different among the cores. The cores are divided into groups, in which each group is a NUMA node. Each NUMA node has its own memory banks. When a core access memory that is located on the same NUMA node, we call this type of access local access. When the core access memory located on other NUMA node, we call the access remote access. We discuss some related work about data mappings for NUMA architectures in Section 2.3.2.

The mapping techniques, in general, can be divided in two main groups: static and dynamic. Static mappings are based on information gathered before the execution of the applications. They require previous analysis of the behavior of the applications to generate a profile that will guide the mapping. This step to generate a profile usually is a time consuming task, requiring simulation or other tools, and relies on information provided by programmers and compilers. On the other hand, dynamic mappings are usually lightweight and do not require previous profiling of the applications. We discuss some related work about static and dynamic mapping in Sections 2.2 and 2.3, respectively.

2.1 Properties of a thread mapping mechanism

To be suitable for a real-world environment, a mechanism to find the communication pattern between threads should present some properties according to CRUZ; DIENER; NAVAUX (2012) and DIENER et al. (2010). Some of them are:

Detect communication pattern and dynamic behavior 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. Furthermore, some applications change their behavior and the communication pattern during the execution. Therefore, 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.

Low impact on performance – The mechanism should have a very low overhead in order not to interfere with the 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 hence is less efficient.

Provide an accurate communication pattern – The detection of the communication pattern should be as accurate as possible to allow a beneficial mapping to be performed. In the case of shared memory applications, this means that the observation should happen as directly as possible by monitoring the memory accesses. 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.

Avoid the false communication problem – False communication can be temporal, as when two threads access the same address, but at different times during the execution. Another scenario of false communication is the classical false sharing problem, which is a spatial false communication, in which a cache line is present in more than one cache; however, the cores are accessing different addresses inside the cache line. Both scenarios should not be considered as communication.

Independence from the implementation of the application – 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, 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.

2.2 Static thread mapping

In this section, we discuss some related work about static mapping. To statically map an application, it must be executed inside controlled environments so that information about the communication pattern can be gathered. This information is used in future executions of the application to map their threads to the cores. There are three main types of controlled environments used to monitor the memory accesses of the applications: emulation, simulation and dynamic binary analysis (DBA).

Emulators are programs that implement some particular instruction set (ISA) entirely by software. They are able to run executable files for the targeted ISA, allowing them to be executed in any machine. Emulators focus on performance and only guarantee that the program output for a given input will be the same of the real machine. Some examples of emulators are Qemu (BELLARD, 2005) and Bochs (SHWARTSMAN; MIHOCKA, 2008). Qemu uses a technique called dynamic binary recompilation, which consists of translating the executable of the the targeted ISA to a binary code compatible to the host machine during runtime. Bochs is based on interpretation and hence is much slower than Qemu. To minimize the high overhead imposed by interpretation, Bochs implements a cache of decoded instructions. The first time an instruction is executed, the operands and the operation are stored in the cache. The future accesses to the same instruction use the information from the decoded cache.

Simulators are also programs that implement some particular instruction set (ISA) entirely by software. They are able to run executable files for the targeted ISA, allowing them to be executed in any machine. Simulators, besides generating the same output of the real machine, also focus on implementing the virtual machine so that the behavior will be similar to the real machine. One example of simulator is Simics (MAGNUSSON et al., 2002). Simics is a full system simulator that achieves a good trade-off between realistic simulation and performance. By doing so, Simics allows real benchmarks to be simulated with a reasonable accuracy compared to a real machine. It allows the simulation of latencies in some components, which makes it possible to simulate the memory hierarchy and pipeline stalls.

Dynamic binary analysis tools enable instrumentation of executable files so that their behavior can be analyzed while running the programs. Some examples of dynamic binary analysis tools are Valgrind (NETHERCOTE; SEWARD, 2007) and Pin (BACH et al., 2010). The feature of Valgrind that is most related to our work is the BBV generator. A Basic Block is a sequence of instructions that contains one entry and one exit points. A Basic Block Vector (BBV) is the set of all Basic Blocks of an application. The tools outputs the BBV of an application in a format compatible with the Simpoint simulator (HAMERLY et al., 2005). Simpoint performs simulation in small steps from different phases of the application. Then, it estimates the global behavior of the application by analyzing the BBV and the results given from each phase. This approach is based on the concept that the behavior of the applications are cyclic.

Valgrind also supports the instrumentation of multi-threaded applications. However, the thread execution is serialized, thereby the behavior of parallel applications is totally different from the original, as only one thread will be executing at a time. On the other hand, Pin does not impose serialization on multi-threaded applications. It provides synchronization primitives, such as locks, so that race conditions can be prevented. The synchronization primitives present high overhead and should be used with precaution. Private storage for each thread is provided by the thread local storage (TLS), which represents an efficient method to store data that must be unique to each thread. Furthermore, Pin has tools to help the analysis of programs based on the most used APIs, such as OpenMP (OPENMP, 2008) and Threading Building Blocks (REINDERS, 2007).

The tools presented in the previous paragraphs are controlled environments that can be used to detect the communication patterns of parallel applications. They demand a lot of computational resources, drastically increasing the overhead of current thread mapping approaches. In the rest of this section, we present some related work about thread mapping that use these controlled environments to gather information to map the threads of parallel applications.

2.2.1 Static thread mapping based on simulation

In BARROW-WILLIAMS; FENSCH; MOORE (2009), a technique to collect the communication pattern between the threads of parallel applications based on shared memory is evaluated. Their method consists of two main steps: generation of memory access traces and the analyses of the traces. To perform the first step, they instrumented Simics to register all the memory accesses in files. The simulated ISA was the *x86*. To obtain more realistic results, a memory hierarchy was also simulated with Simics while generating the traces. The memory hierarchy had private L1 caches and one large shared L2 cache using a coherence protocol. In the second step, the memory traces are analyzed to determine the communication pattern of the applications.

To consider an access to the memory as communication between the threads, the access should fulfill three criteria. The first one is that more than one thread must access the same address. The second criteria is that when one thread performs several consecutive writes to the same address, only the last access represents a communication. The third is that when one thread performs several consecutive reads to the same address, only the first access represents a communication. These last two criteria are necessary to ignore the memory access that occurred due to register number constraints. This happens when several consecutive operations must be performed in one variable, but the architecture does not provide enough registers to hold all the involved variables.

The authors categorize the memory accesses in three types: read-only, migratory and producer-consumer. Read-only accesses are the ones that only read accesses are performed after the initialization of the variable. Migratory accesses are represented by atomic read-write operations. Producer-consumer is a pattern that one data is written by one thread, the producer, and then is read by another thread, the consumer. However, as few memory accesses fulfill this definition, the authors used a less strict rule to consider a memory access as a producer-consumer. They extended the definition to allow an arbitrary number of producers, and to consider a thread as a consumer when the it reads the data for at least 50% of the performed writes.

The technique was evaluated using the *Splash-2* (WOO et al., 1995) and *Parsec* (BIENIA et al., 2008) benchmarks. These benchmarks were chosen because they cover different targets: *Splash-2* is more suitable to evaluate clusters, while *Parsec* is focused in multicore architectures. The analysis of the results showed that 1.5% of the read accesses represent communication. For the write accesses, 4.2% and 20.4%, respectively for *Splash-2* and *Parsec*, represent communication. As the main goal of the work was just to characterize the communication pattern, the authors did not ran any performance tests using the collected data.

The work described in DIENER et al. (2010) also uses Simics to discover the communication pattern of parallel applications. However, the authors used the collected information to map the threads to cores according to the amount of communication. They developed two algorithms to map the threads. The first one tries every possible mapping and selects the one that maximizes the amount communication between the threads. The problem of this approach is that the time complexity is exponential, hence it is viable only for small number of threads. The second algorithm is a greedy heuristic and presents polynomial time complexity. The applications were also executed using the original scheduling policy of the operating system.

The authors evaluated their proposal by mapping a subset of the *Splash-2* and *Parsec* benchmarks on the Intel Nehalem architecture. Performance was improved by up to 45% when compared to the native scheduler of the Linux operating system. Thread migration imposed by the Linux scheduler requires context switches, which increases the number of cache misses and leads to performance degradation when transferring the data to other caches (MOGUL; BORG, 1991). In CRUZ; ALVES; NAVAUX (2010), it is shown that, in some situations, thread migration can harm the performance more than any random thread mapping.

2.2.2 Static thread mapping based on dynamic binary analysis

The work presented in BIENIA; KUMAR; LI (2008) uses the Pin tool to collect information about the applications. They analyze the communication pattern and the distribution of the instructions of the applications by their type. For the communication pattern, they evaluate the amount of memory shared by the threads, separating read-only data and data that suffered rewrites. Furthermore, they counted the number of accesses to regions of memory that are shared, both write and read accesses. The working set (TANENBAUM, 2007) of the applications were also taken into account. The size of the working set was estimated by making use of the cache miss rate, since the amount of cache misses is usually proportional to the size of the working set.

The results show that, even for small cache memories of $1mb$, the miss rate is lower than 1.25% for the Parsec benchmark. To decrease the cache miss to almost 0%, the cache size was set to $128mb$. This indicates that, despite the cache miss rate being proportional to the size of the working set, the scale is not direct. Another interesting result of the work was the ratio of write accesses to shared cache lines. This is an important measure because write operations performed to shared cache lines generate invalidation or update messages by cache coherence protocols. With the size of current last level caches, which are around $32mb$, less than 6% of the memory accesses result in writes to shared cache lines. This number rises to around 7.5% when considering a $128mb$ cache size. As the goal of the paper was just to analyze the behavior of the applications, no performance tests were realized. However, these results suggest that obtaining performance improvements with thread mapping is challenging, since the amount of private data overwhelms the amount of shared data in current parallel applications.

2.2.3 Static thread mapping based on hardware counters

Hardware counters present on current architectures may also be used to guide the static thread mapping. In OTT et al. (2008), hardware counters were employed in order to measure the quality of a specific thread mapping. The communication among the threads of the applications was not considered, since hardware counters present on current architectures do not provide this kind of information. Instead, they used the million instructions per second (MIPS) as metric. This metric indirectly estimates the quality of the mapping, since it is expected that the value of the MIPS to be higher for better mappings. Each application was executed with every possible mapping, and the one that outputs the highest MIPS was chosen.

To evaluate their proposal, they used their mechanism to map applications from the *Spec OMP* benchmark (SAITO et al., 2002). The authors claim that their method was able to find the best mapping for all the applications. However, they did not explain how they obtained the best mapping. Furthermore, the results are not conclusive and the manner they were presented lacks organization, which makes them difficult to be interpreted. Nevertheless, it is interesting that they used hardware counters, which requires less computational resources than simulation and dynamic binary analysis and could be implemented directly on the operating system scheduler.

2.2.4 Static process mapping applied to message passing based applications

As already mentioned, the detection of the communication pattern of message passing based applications is straightforward compared to shared memory based applications. In RODRIGUES et al. (2009), a technique to statically map parallel applications based on the

Message Passing Interface (MPI) (MPI, 2009) is proposed. To detect the communication pattern, the authors created wrappers to the MPI primitives that monitors the source and destination fields of each message, as well as the amount of exchanged data between them. With these information, they generate a complete graph in which the vertices represent the processes and the edges the amount of communication.

To map the processes on the processing units they used a tool called *Scotch* (SCOTCH, 2010). *Scotch* requires as input two graphs: one to describe the application and other to describe the machine. The graph of the application is generated as explained in the above paragraph. In the graph of the machine, the vertices represent the processing units and the edges the bandwidth of the link that connects them, if there is one. *Scotch* maps the application graph to the machine graph by applying heuristic algorithms, most of them based on graph partitioning. The generated mapping tries to minimize the overhead imposed by the communication while keeping the load balancing. The bandwidth of the links of the target machines were measured using a ping-pong application, which just sends messages between all the processing units and reports the time took to transmit them.

They used the weather forecast program BRAMS (BRAMS, 2009) to evaluate their proposal. They obtained up to 9% of performance improvement, and a reduction of 20% on the time spent waiting for messages. The mapping also resulted in a reduction in the network traffic between the nodes of the cluster. One weak point of this work is that the only metric tested was the amount of exchanged data. As stated in CHEN et al. (2006), this metric is appropriate for applications in which the size of the messages far surpass the number of messages. On the other hand, if the message sizes are too small, the network latency dominates the transmission time, which makes the number of messages metric more suitable than the amount of data.

2.3 Dynamic thread mapping

In this section, we discuss some related work about dynamic mapping.

2.3.1 Dynamic thread mapping using hardware counters

The works described at TAM; AZIMI; STUMM (2007a,b); AZIMI et al. (2009) show that hardware performance counters already present in current processors may be used to dynamically map parallel applications. They schedule threads by making use of hardware counters present in the Power5 processor. Four main steps are performed: monitoring, discovery of the communication pattern, thread clustering and migration. The monitoring step consists of checking the hardware counter that stores the amount of cycles that the core was stalled due to accesses to remote cache memories. When the stall time exceeds a certain threshold, the mapping mechanism is enabled. This threshold is employed to reduce the overhead.

When the mapping mechanism is enabled, the first procedure is to detect the communication pattern. This was accomplished by reading a hardware counter of the Power5 that stores the latest address that resulted in a remote cache access. A list of addresses is kept for each core. After some time, when the size of the lists are large enough, the thread clustering step is initiated. If the same address is present in more than one list, that means that this address is shared between the corresponding cores and is used for communication. Therefore, the thread clustering step consists of searching the lists of addresses of each core for matches. Finally, the threads migrate to cores so that the number of remote access is reduced.

Performance was increased by up to 7% and the number of memory accesses to remote cache memories was reduced by up to 70%. This work has three main disadvantages. The first is that the steps of discovering the communication pattern present high overhead, because it requires too much traps to the kernel. The thread clustering step is also expensive, but as it is not performed so frequently, it does not have a high impact. The second disadvantage is a result of the first one. As two of the steps have high overhead, the mapping mechanism is enabled occasionally, which reduces the accuracy of the results. The third disadvantage is that only the remote cache accesses are monitored, hence it is not possible to detect communication among all the threads.

In BROQUEDIS et al. (2010), a library called ForestGOMP is introduced. This library integrates into the OpenMP (OPENMP, 2008) runtime environment and gathers information about the different parallel sections of the applications from hardware performance counters. The library generates data and thread mappings for the regions of the application. As the library performs data mappings, it focus on NUMA architectures. The library tries to keep the threads that communicate nearby according to the memory hierarchy, as well as to place the memory pages in NUMA nodes close to the cores that access the page. As stated in TERBOVEN et al. (2008), Linux first-touch policy consists of mapping a page to the NUMA node of the core that first access the page, and is not efficient due to the different behavior of parallel applications in different phases of the program. ForesGOMP adds a next-touch policy, in which the pages migrate to the NUMA node of the next core that access the page. The library also supports the migration of individual pages.

They improved performance by up to 11.6% using the NAS parallel benchmarks. They also evaluated the performance with a benchmark called *Stream* (STREAM, 2011), which measures the memory bandwidth, obtaining up to 80% of improvement. This work has two disadvantages. The first one is that the hardware counters they used to guide the thread and data mapping only indirectly estimate the communication patterns. The second major problem is that their work is limited to OpenMP based applications. Applications based on other APIs, such as Intel TBB or Posix Threads, are not able to benefit from their library.

2.3.2 Dynamic data mapping applied to NUMA machines

The ForestGOMP library (BROQUEDIS et al., 2010), introduced on the previous section (Section 2.3.1), already performs data mappings for NUMA machines. Another interesting work is described at AWASTHI et al. (2010). They have developed page migration mechanisms that uses the load balancing between the memory controllers and the row-buffer hit rate as main metrics. The reason for optimizing the load balancing is straightforward, as it will cause a better distribution of the memory traffic across the NUMA nodes. Optimizing the row-buffer hit rates, on the other hand, is very interesting. If you have high row-buffer hit rates, it suggests that the same data is being re-used. It is not an accurate information because access to different pages may also result in a row-buffer hit, but, at least, a miss in the row-buffer only happens if a different page is accessed. As a result, the row-buffer hit rate indirectly estimates the amount of shared pages between the cores that are accessing a NUMA node.

Two page migrations mechanisms were developed. The first is called Adaptive First-Touch and consists of gathering statistics of the memory controller to map the data of the

Table 2.1: Summary of the characteristics of current thread mapping mechanisms.

Method	**Detect communication pattern and dynamic behavior during execution**	**Impact on performance**	**Accuracy of the communication pattern**	**Avoid the false communication problem**	**Independent from the implementation of the application**
Static, using simulation	No	High	High	Yes	No
Static, using DBA	No	High	High	Yes	Yes
Static, using hardware counters	No	Low	Low	No	Yes
Dynamic, using hardware counters	Yes	Low	Low	No	Yes

application in future executions. However, this mechanism fails if the behavior changes among the different phases of the application. The second mechanism uses the same information, but allows dynamic page migration during the execution of the application. In their heuristic, a page must migrate when the row-buffer hit rate drops by 10%. They select the destination NUMA node considering the difference of the access latency between the source and destination NUMA nodes, as well as the row-buffer and the load of the destination memory controller.

Their mechanisms were implemented in the Simics simulator, leading to a performance improvement of up to 35%. The major problem of this work is that the information about the communication pattern between the threads, as well as which data each thread is using, is unreliable, since it is based only in the row-buffer hit rate. The page to be migrated is randomly chosen, and may lead to an increase on the number of remote accesses. Additionally, threads that access the same memory pages may be executed on different NUMA nodes, since they do not map threads that communicate on the same node. This also leads to an increase of the number of remote accesses.

2.3.3 Dynamic process mapping applied to message passing based applications

In SONNEK et al. (2010), virtual machines running on clusters are migrated among the different nodes considering the amount of communication between them. They detect the communication between the virtual machines by monitoring the source and destination fields of the packets sent on the network. By dynamic migrating the virtual machines to nearby nodes of the cluster, they improved performance by up to 42% and reduced the network communication cost by up to 85%.

2.4 Summary of the state-of-art

In this chapter, we presented the state-of-art related to mapping techniques. Thread mapping, data mapping and process mapping mechanisms were evaluated. One first conclusion is that process mapping of parallel applications based on message passing are rather straightforward, at least from the point of view of the computer architecture. As the communication pattern is easily discovered by monitoring the source and destination of the messages, the problem is reduced to determine which core will execute each process. This is more of a mathematical and combinatorial optimization problem than a computer architecture problem. On the other hand, thread mapping of parallel applications based on shared memory imposes a real challenge on computer architecture, since it is very difficult to accurately detect the communication pattern without requiring time consuming tasks such as profiling.

Table 2.1 summarizes the characteristics of current thread mapping mechanisms. We can categorize current thread mapping mechanisms in two groups: the mechanisms that impose high overhead, but generate an accurate communication pattern, and the mechanisms that are light-weight, but rely on indirect and unreliable information about the communication. To our knowledge, there is no solution that was able to accurately detect the communication pattern of shared memory based parallel applications with a low overhead. Our goal is to fulfill this gap. We developed mechanisms that dynamically generate accurate communication patterns with a low overhead for parallel applications based on shared memory. Our mechanisms also allow the thread mapping to be performed dynamically by the operating system during the execution of the applications. Furthermore, they do not depend on any particular parallel programming library or any kind of modifications to the source code of the applications.

3 PROPOSED METHODS FOR DYNAMIC DETECTION OF THE COMMUNICATION PATTERN

In this chapter, we explain our proposed mechanisms for dynamic detection of the communication pattern. The proposals can be classified in two main categories: cache coherence based and TLB based. The communication can be analyzed by grouping different number of threads. To calculate the amount of communication between groups of threads of any size, the time and space complexity raises exponentially. Therefore, the communication was evaluated only between pairs of threads, generating a communication matrix. Although this may 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.

3.1 Exploiting Cache Coherence Protocols

Cache coherence protocols are responsible for keeping data integrity in architectures where more than one cache memory is present, as is common in multicore and multiprocessor environments. These protocols keep information about whether a line is private or shared between two or more caches. This can be exploited in order to estimate the amount of communication between the threads, since an access to a line shared by two or more caches represents a communication. Small modifications to the protocols and the hardware are required to identify communication patterns. We used the MESI protocol (STALLINGS, 2006) as base for our cache coherence based mechanisms, and they can be adapted to work with any protocol, such as MOESI.

In MESI based protocols, an invalidation message is sent when a write is performed in a shared or invalid cache line. However, no message is sent when a read is performed in a shared cache line. Hence, we detect the communication when the cache lines are invalidated. Nevertheless, read transactions are considered when an invalidation message arrives, because cache lines are invalidated regardless of their state. This decision was taken considering that write operations have a greater impact on performance than read operations (CHISHTI; POWELL; VIJAYKUMAR, 2005).

There are two main types of cache coherence protocols: snoop and directory. We propose modifications to both snoop and directory protocols in the rest of the section. The examples and figures were based on a machine which consists of four cores.

3.1.1 Snoop Protocols

In this section, we explain how to detect the communication pattern in snoop protocols. In these protocols, each cache is responsible for keeping the state of its cache lines.

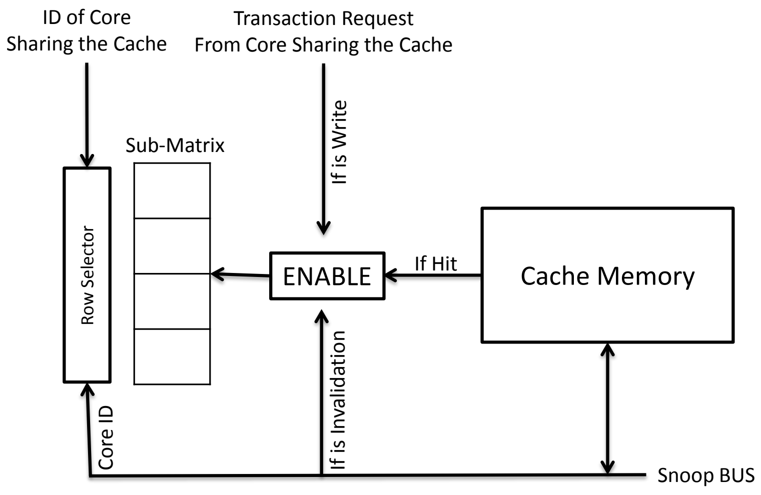


Figure 3.1: Snoop based mechanism for private caches.

Since snoop protocols usually do not keep any information about which caches are sharing each line, the coherence messages are broadcasted to all caches (EGGERS; KATZ, 1989). Snoop protocols can be used with caches that are private to one core and with caches that are shared between two or more cores. We propose different mechanisms for private and shared caches.

3.1.1.1 Mechanism for Private Caches

Our first approach to detect the communication pattern in snoop protocols is appropriate for private cache memories. We add a matrix to each cache to store the communication corresponding to the core that is connected to the cache, therefore the matrix consists of only one column. The number of rows of this matrix is equal to the total number of cores in the system. From now on, we will call this matrix *sub-matrix*, since it is part of the communication matrix. The procedure to obtain the communication matrix is explained in Section 3.1.3.1. In order to identify which cores are communicating, the id of the core that sent the coherence message is also required. To provide this information, the id of the core must be sent along with the message. This scheme is not suitable for shared caches, since we do not know which of the cores that share the cache have accessed each line.

Figure 3.1 shows how the mechanism works. Table 3.1 presents some examples, considering that all transactions happens on the same cache line. In the Table, the letters *M*, *E*, *S* and *I* represent the coherence state. When a write is performed in a shared cache line, as in Example 1, we send the invalidation message along with the id of the core that sent the message in the snoop bus. Those caches that invalidate a line in response to this message increment their sub-matrix by one in the cell whose row is the core id from the snoop bus. The same happens when a write is performed in an invalid cache line, as in Example 2. Example 3 illustrates how read transactions are taken into account when an invalidation arrives.

The space required to store each sub-matrix is $P \cdot C$, where P is the number of processing cores of the machine and C is the size of each cell of the sub-matrix in bytes. As there are P caches, the total space used to store the matrices is $P^2 \cdot C$ bytes.

Table 3.1: Examples of the snoop based mechanism for private caches.

Ex.	Core 0 Cache 0	Core 1 Cache 1	Core 2 Cache 2	Core 3 Cache 3	Action
1	S	I	S	S	Core 0 Writes Cache 0 sends invalidation
	M	I	I	I	Cache 2 updates its sub-matrix in cell 0-0 Cache 3 updates its sub-matrix in cell 0-0
2	I	E	I	I	Core 0 Writes Cache 0 sends invalidation
	M	I	I	I	Cache 1 updates its sub-matrix in cell 0-0
3	I	I	I	I	Core 0 Reads
	E	I	I	I	Core 2 Reads
	S	I	S	I	Core 3 Writes Cache 3 sends invalidation
	I	I	I	M	Cache 0 updates its sub-matrix in cell 3-0 Cache 2 updates its sub-matrix in cell 3-0

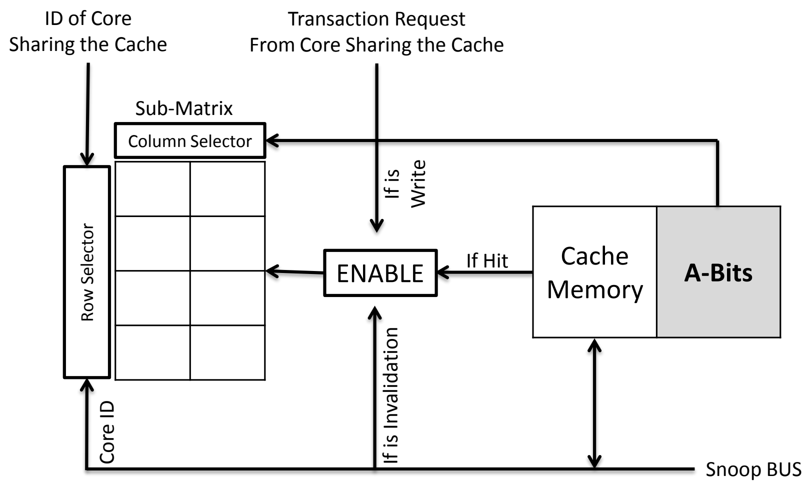


Figure 3.2: Snoop based mechanism for shared caches.

3.1.1.2 *Mechanism for Shared Caches*

The previous protocol (Section 3.1.1.1) requires small modifications to the hardware, but only works for private caches. Its constraint is that it does not determine which core accessed a given cache line when there is more than one core sharing a cache. The protocol described in this section overcomes this issue, but requires more modifications to the original hardware.

To make it possible to use the mechanism with shared caches, we add one access bit (A-Bit) per core sharing a cache to every cache line. These access bits show which of the cores sharing a cache accessed a cache line. We add a sub-matrix to each cache to store the amount of communication correspoding to the cores that are sharing the cache. The number of rows of the sub-matrix is equal to the total number of cores, and the number of columns is equal to the number of cores sharing the cache.

Figure 3.2 shows how the mechanism works, considering that there are 2 cores sharing each cache. Therefore, 2 A-Bits are added to each cache line, and each sub-matrix has 2 columns. Table 3.2 present examples of the operation of the mechanism. C_x is the coherence state of cache x and A_x is the Access Bit corresponding to core x. It is important to

Table 3.2: Examples of the snoop based mechanism for shared caches.

Ex.	Core 0-1 Cache 0			Core 2-3 Cache 1			Action
	C0	A0	A1	C1	A2	A3	
1	I	0	0	I	0	0	Core 0 Reads Cache 0 Sets A0
	E	1	0	I	0	0	
2	E	1	0	I	0	0	Core 1 Writes Cache 0 Sets A1 Cache 0 updates its sub-matrix in cell 1-0
	M	1	1	I	0	0	
3	S	1	0	S	0	1	Core 1 Writes Cache 0 Sets A1 Cache 0 updates its sub-matrix in cell 1-0 Cache 0 sends invalidation
	M	1	1	I	0	0	Cache 1 updates its sub-matrix in cell 1-1
4	S	1	0	S	1	1	Core 1 Writes Cache 0 Sets A1 Cache 0 updates its sub-matrix in cell 1-0 Cache 0 sends invalidation
	M	1	1	I	0	0	Cache 1 updates its sub-matrix in cells 1-0 and 1-1

mention that A_2 and A_3 point to columns 0 and 1 of the sub-matrix, since each sub-matrix only stores the communication corresponding to the cores that share them.

When a memory request arrives on the cache, the cache sets the A-Bit corresponding to the core that initiated the transaction on the requested cache line, as in Example 1. If a write hit occurs, the sub-matrix of the accessed cache is incremented in the cells whose row is the id of the core sharing the cache that initiated the transaction, and columns where the A-Bits are set (Example 2). This is done in order to detect the communication among the cores that share the cache. Besides, if the write hit happens in a shared cache line, an invalidation message is broadcasted along with the id of the core that initiated the transaction. Those caches that invalidate a line in response to this message increment their sub-matrix by one in the cells whose row is the core id from snoop bus, and columns where the A-Bits are set, as depicted in Example 3. It is important to note that, when more than 1 A-Bit is set, more than one cell is updated, as in Example 4. The invalidation is also broadcasted along with the id of the core when a write is performed in an invalid cache line.

The space required to store each sub-matrix is $P \cdot S \cdot C$, where P is the number of cores of the machine, S is the number of cores sharing the cache and C is the size of each cell of the sub-matrix in bytes. The overhead of the A-Bits for each cache is $\frac{B \cdot S}{L}$ bits, where B is the size of the cache in bytes, L is the size of each cache line in bytes, and S is the number of cores sharing the cache. As there are $\frac{P}{S}$ caches, the total space used to store the matrices is $P^2 \cdot C$ bytes and the total space required for the A-Bits is $\frac{B \cdot P}{L}$ bits. Since one byte contains 8 bits, the space complexity is $P^2 \cdot C + \frac{B \cdot P}{8 \cdot L}$ bytes.

3.1.2 Directory protocols

In this section, we explain how to detect the communication pattern using directory cache coherence protocols. In directory protocols, the directory stores the state of the cache lines of all caches and also keeps track of which caches contain each line. This can be seen in the Example 1 of Table 3.3. Coherence messages are sent only to the caches

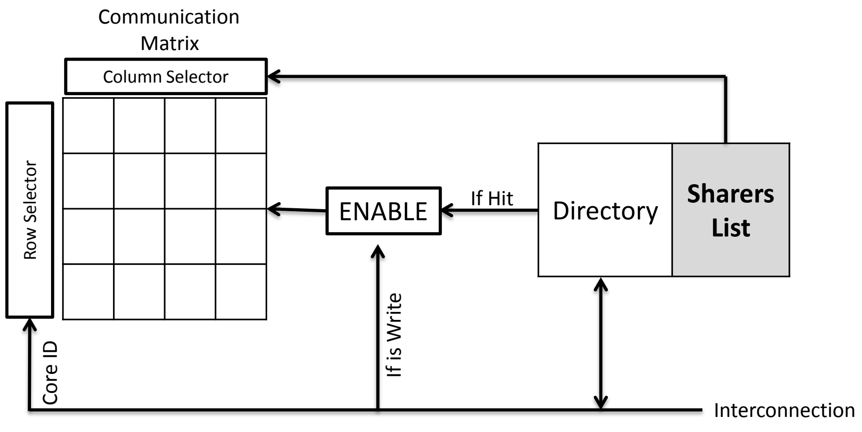


Figure 3.3: Centralized directory based mechanism.

Table 3.3: Examples of the centralized directory based mechanism.

Ex.	**Core 0**	**Core 1**	**Core 2**	**Core 3**	**Directory**				**Action**
	Cache 0	**Cache 1**	**Cache 2**	**Cache 3**	**S0**	**S1**	**S2**	**S3**	
1	I	I	I	I	0	0	0	0	Core 0 Reads Directory sets Exclusive to Cache 0
	E	I	I	I	1	0	0	0	Core 2 Reads Directory sets Shared to Cache 2 Directory sets Shared to Cache 0
	S	I	S	I	1	0	1	0	
2	S	I	S	I	1	0	1	0	Core 1 Writes Directory sends invalidation to Caches 0 and 2 Directory updates its sub-matrix in cells 1-0 and 1-2
	I	M	I	I	0	1	0	0	
3	S	S	S	I	1	1	1	0	Core 1 Writes Directory sends invalidation to Caches 0 and 2 Directory updates its communication matrix in cells 1-0 and 1-2
	I	M	I	I	0	1	0	0	

that contain the corresponding line, hence there is no need for broadcasts (AGARWAL et al., 1988). We will refer to the structure that keeps the caches that share a line as *sharers list*. Likewise, we will call the caches that are sharing a given cache line as a *sharer* of the line.

It is important to note that the directory keeps track of which caches are sharing each line, not which cores are sharing each line. Therefore, we have to use the information about the last private cache level to detect if the corresponding core shares some line, since shared cache levels can be accessed by more than one core. For instance, considering an architecture with 2 cache levels, and just the L1 cache is private, the directory is not able to tell which core has accessed each line of the L2 cache. However, the lines present on the L1 cache were surely accessed by the corresponding core.

Different modifications to the hardware were proposed for centralized and distributed directories.

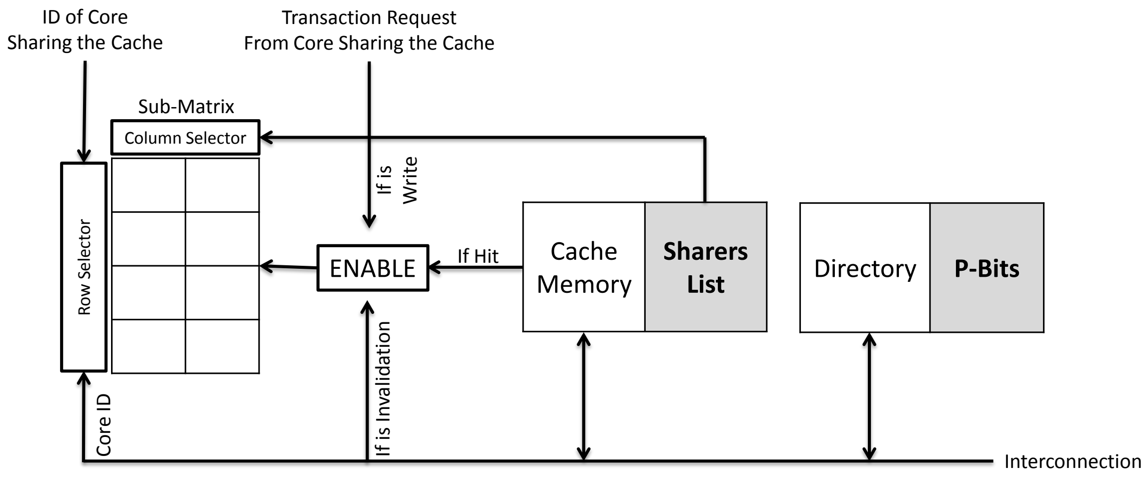


Figure 3.4: Distributed directory based mechanism.

3.1.2.1 Mechanism for Centralized Directories

In centralized directories protocols, there is only one directory. Therefore, all the coherence information is found on this directory. To allow the detection of the communication pattern, the main modification required to the hardware is the addition of the communication matrix to the directory. The number of rows and columns of the communication matrix are equal to the total number of cores. In the original protocol, when a core requests a write transaction, if the cache line is shared or invalid, it access the directory to check which caches are sharing the requested cache line. Then, the directory sends invalidation messages to all the caches that share the corresponding cache line and clears the sharers list. After that, the directory set as sharer only the caches that are shared by the core that requested the write transaction.

Our mechanism is depicted in Figure 3.3. Table 3.3 presents examples of how the mechanism works, in which S_x is the Sharer Bit corresponding to cache x. To detect the communication pattern, before clearing the sharers list, our mechanism increment the cells of the communication matrix whose row is the id of the core that initiated the transaction, and columns corresponding to the sharers list. Examples 2 and 3 demonstrate this behavior.

The space required to store the communication matrix is $P^2 \cdot C$ bytes, where P is the number of cores of the machine and C is the size of each cell in bytes.

3.1.2.2 Mechanism for Distributed Directories

Distributed directory based protocols can be implemented in different ways. In one implementation suitable for multi-core architectures, the directory keeps track of which processors are sharing each line (ZHAO; SHRIRAMAN; DWARKADAS, 2010). The directory only forwards the request to the processors that have the corresponding cache line. It is responsibility of each processor to keep track of which caches inside them hold the requested cache line. Therefore, there are 2 types of sharers list: one that is kept by the directory and points to processors, and one that is kept by each processor and points to its caches. The main modification required to the hardware is the addition one sub-matrix to each processor. The number of rows of the sub-matrix is equal to the total number of cores, and the number of columns is equal to the number of cores sharing the cache.

In the original protocol, when a core requests a write transaction, if the cache line

Table 3.4: Examples of the distributed directory based mechanism.

Ex.	Core 0-1 Cache 0			Core 2-3 Cache 1			Directory		Action
	C0	S0	S1	C1	S2	S3	P0	P1	
1	E	1	0	I	0	0	1	0	Core 1 Writes Cache 0 Sets S1 Cache 0 updates its sub-matrix in cell 1-0
	M	1	1	I	0	0	1	0	
2	S	1	1	S	1	1	1	1	Core 1 Writes Cache 0 updates its sub-matrix in cell 1-0 Cache 0 access the directory
	M	1	1	S	1	1	1	1	The directory sends an invalidation to Cache 1 The directory clears the bit P1
	M	1	1	I	0	0	1	0	Cache 1 updates its sub-matrix in cells 1-0 and 1-1

is shared or invalid, it accesses the directory to check which processors are sharing the requested cache line. Then, the directory sends invalidation messages to all the processors that share the corresponding cache line and clears its sharers list. The processors that receive the invalidation messages check which caches are sharing the corresponding line, and then forward the invalidations to them. After that, the directory set as sharer only the processor that requested the write transaction. The processor set as sharer only the caches that are shared by the core that requested the write transaction.

Our mechanism works as in Figure 3.4 and Table 3.4 contains some examples, considering that there are 2 cores sharing each cache, and that each pair of cores belong to different processors. In the Table 3.4, C is the coherence state, S_x is the Sharer Bit corresponding to cache x and P_x is the bit that represents if processor x holds the corresponding line. It is important to mention that S_2 points to column 0, and S_3 to column 1, since each sub-matrix only stores the communication relative to the cores that share them. When a write occurs in an exclusive or modified lines, there is no need for accessing the directory. In this case, we only update the sub-matrix of the corresponding cache in the cells whose row is the id of the core that initiated the transaction, and columns corresponding to the sharers list. This procedure is demonstrated in Example 1.

When the write is performed in a shared or invalid cache line, the cache, besides updating its own sub-matrix, also has to access the directory. This has to be done in order to send invalidation messages to the other processors that have the corresponding cache line. When the invalidation that was forwarded by the directory arrives at the processor, before forwarding the invalidation to the caches that share the cache line, our mechanism increment the cells of the communication whose row is the id of the core that initiated the transaction, and columns corresponding to the sharers list. Example 2 demonstrates this procedure.

The space required to store each sub-matrix is $P \cdot S \cdot C$, where P is the number of cores of the machine, S is the number of cores sharing the cache and C is the size of each cell in bytes. As there are $\frac{P}{S}$ caches, the total space used to store the matrices is $P^2 \cdot C$ bytes.

3.1.3 Properties and procedures common to all cache coherence based mechanisms

In this section, we present properties and procedures that are common to all the proposed cache coherence based mechanisms. First, we explain how to obtain the amount of communication between the threads from the matrices kept by our mechanisms. Afterwards, we show the common hardware requirements and the software overhead.

3.1.3.1 Obtaining the amount of communication

For the mechanism based on centralized directories, obtaining the amount of communication between two threads happens as follows. Considering i and j identifiers of cores, we only have to add the cells (i, j) and (j, i) of the communication matrix. This addition is necessary because we want to know the communication between the two threads regardless of the direction of the communication.

Regarding the other mechanisms, the communication matrix is divided in sub-matrices attached to the caches. Each sub-matrix contains the amount of communication corresponding to the cores that are sharing the cache. If we consider that the columns of all sub-matrices are virtually concatenated, sorted by the corresponding core id, we obtain a square matrix. This square matrix is the communication matrix, and the procedure to obtain the amount of communication between two threads is the same as in centralized directories.

3.1.3.2 Common Hardware Requirements

All mechanisms require one adder unit per matrix to increment its cells. However, more than one cell of the matrix may be incremented, as it happens when the number of columns of each sub-matrix is greater than one. In these cases, the number of adder units per sub-matrix, as well as read and write ports, must be equal to the number of columns of the sub-matrix to allow the increments to be performed simultaneously. Otherwise, the increments are serialized. Additionally, instructions must be added to the instruction set to allow the operating system to read and write to the communication matrix.

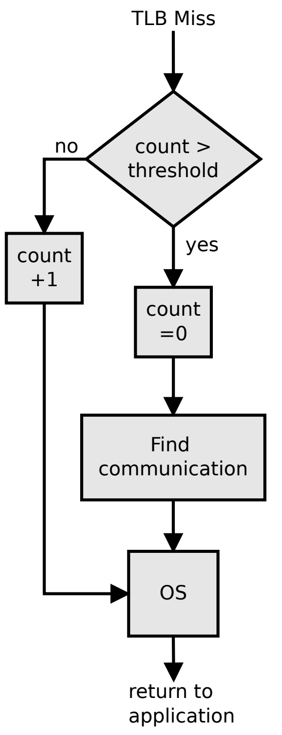
3.1.3.3 Software Overhead

The communication pattern is detected entirely by hardware. The only procedure required from the software is the addition of two cells of the communication matrix, as explained in Section 3.1.3.1.

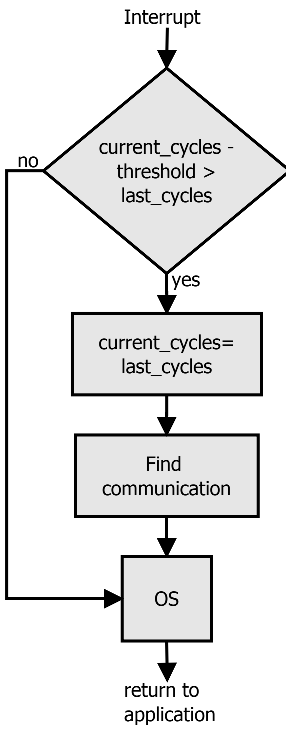
3.2 Exploiting the Translation Lookaside Buffer

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 multicore 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



(a) Flowchart for the software-managed TLB.


Interrupt
current_cycles - threshold > last_cycles
no
yes
current_cycles= last_cycles
Find communication
OS
return to application

(b) Flowchart for the hardware-managed TLB.

Figure 3.5: Flowcharts for the proposed TLB based mechanisms.

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 multicore 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 discover the communication pattern. Therefore, we describe our proposed mechanism separately for each of the two TLB management types.

3.2.1 Mechanism for Software-Managed TLBs

In some RISC architectures, such as SPARC (SPARC, 2000) and MIPS (MIPS, 1996), 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 is 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 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 3.5(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 of 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 $\Theta(P \cdot E)$, where P, is the number of processing cores and E is the number of entries of the TLB. The complexity increases linear with P since we have to check all the other TLBs, and it is also linear with E because all the entries of the TLB are searched for matches. However, considering a set associative TLB, the time complexity 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. The space complexity to store the mirrors of the TLBs in the main memory is $P \cdot E \cdot T$ bytes, where T is the size of each entry of the TLB in bytes. To store the communication matrix, $P^2 \cdot C$ bytes are required, where C is the size of each cell of the communication matrix in bytes. Therefore, the space complexity is $P \cdot E \cdot T + P^2 \cdot C$ bytes.

3.2.2 Mechanism for Hardware-Managed TLBs

Architectures such as x86 and x86-64 (INTEL, 2004, 2009; AMD, 2007) 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 invalidating the 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. The accuracy and overhead of this mechanism depend on the time between searches.

Our implementation of this proposal for the hardware-managed TLB is presented in the flowchart in Figure 3.5(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

Table 3.5: Comparison between the proposed mechanisms.

Proposal	Hardware Modifications	Time Complexity	Space Complexity	Granularity
Snoop Private	Communication Matrix, Adder unit, and the addition of an instruction to allow the operating system to access the communication matrix	$\Theta(1)$	$P^2 \cdot C$ bytes	Cache Line
Snoop Shared	A-Bits, Communication Matrix, Adder Unit and the addition of an instruction to allow the operating system to access the communication matrix	$\Theta(1)$	$P^2 \cdot C + \frac{B \cdot P}{8 \cdot L}$ bytes	Cache Line
Centralized Directory	Communication Matrix, Adder Unit, and the addition of an instruction to allow the operating system to access the communication matrix	$\Theta(1)$	$P^2 \cdot C$ bytes	Cache Line
Distributed Directory	Communication Matrix, Adder Unit, and the addition of an instruction to allow the operating system to access the communication matrix	$\Theta(1)$	$P^2 \cdot C$ bytes	Cache Line
Software-managed TLB	None	$\Theta(P)$	$P \cdot E \cdot T + P^2 \cdot C$ bytes	Memory Page
Hardware-managed TLB	Addition of an instruction to allow the operating system to access the TLB	$\Theta(P^2 \cdot E)$	$P \cdot E \cdot T + P^2 \cdot C$ bytes	Memory Page

P: number of processing cores
E: number of entries of the TLB
C: size of each cell of the communication matrix in bytes
B: size of a cache in bytes
L: size of each cache line in bytes
T: size of each entry of the TLB in bytes

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 $\Theta(P^2 \cdot E^2)$, where P is the number of processing cores and E is the number of entries 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 E. However, if we use a set associative hardware-managed TLB, the time complexity is decreased to $\Theta(P^2 \cdot E)$, since it is not necessary to check all the entries of the TLB for matches, but just the entries that are on the same set. The space complexity to store the mirrors of the TLBs in the main memory is $P \cdot E \cdot T$ bytes, where T is the size of each entry of the TLB in bytes. To store the communication matrix, $P^2 \cdot C$ bytes are required, where C is the size of each cell of the communication matrix in bytes. Therefore, the space complexity is $P \cdot E \cdot T + P^2 \cdot C$ bytes.

3.3 Summary of the proposed methods

In this section, we compare our proposed mechanisms for dynamic discovery of the communication patterns. As outlined in Section 2.1, a mechanism to discover communication patterns between threads should have several properties. As our mechanisms are performed entirely by the hardware and the operating system, they do not depend on the parallelization API and do not require any modification to the application. Moreover, the communication pattern is discovered during the execution time of the application. Regarding the dynamic behavior of applications, our mechanisms provide a good solution for detecting changes in the behavior because the number of possible entries in the TLB and cache is quite low. Data that is not accessed anymore will have its corresponding entry overwritten after a short time and 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 and cache entries.

Table 3.5 summarizes the characteristics. All mechanisms except the software-management TLB need modifications to the current hardware. The snoop mechanism for shared caches is the one that needs more modifications, since it requires the addition of the A-Bits. The TLB mechanisms are more affected by the false sharing problem than the cache coherence ones, since the size of the memory page is much bigger than the size of the cache line. Regarding the time complexity, all the cache coherence based mechanisms execute in $\Theta(1)$, since they operate entirely by hardware. On the other hand, the TLB based mechanisms require the operating system to search for matches among the TLBs of different cores. Regarding the space complexity, it is more relevant for the cache coherence based mechanisms, because the data is stored in the cache system, while in the TLB based mechanisms the data is stored in the main memory.

In our cache coherence based mechanisms, there is one main difference between the mechanisms based on directory and snoop. In the directory, whenever a cache no longer holds some cache line, it is mandatory that it leaves the corresponding sharers list in the directory. This happens because the control of the sharers list is inherent to the directory, which takes this action to prevent unnecessary invalidation messages. However, in our snoop based mechanism, the A-Bits are used only by our mechanism, the coherence protocol does not depend on them. Therefore, we chose to clear the A-Bit corresponding to some core only when an invalidation arrives. Line evictions due to replacements do not clear the A-Bits of the corresponding cores on other caches.

We made this decision because some results indicated that, with low cache sizes, some shared data do not stay enough time on the cache to be considered as shared by the coherence protocol. For instance, consider a system with 2 levels of cache. The L1 caches are private, while the L2 caches are shared by two cores. When a core access some data, the corresponding cache line will be stored by both the L1 and L2 caches of the core. In directory based protocols, if the L1 cache evicts that same cache line, the corresponding core will not be considered a sharer of that cache line anymore. However, in our snoop based mechanism, if the L1 cache evicts some cache line, the A-Bit corresponding to that core (the L1 cache is private and we know which core is related to it) keeps set on the L2 cache. Since the L1 cache size is expected to be much lower than the L2 cache size, the directory based protocols are much more sensitive to the cache size than the snoop based protocols.

Regarding multithreaded cores, such as in simultaneous multithreading (SMT) (TULLSEN; EGGERS; LEVY, 1995), the cache coherence based approaches would require more modifications to the hardware than the ones explained in this thesis. In the snoop based protocols, the A-Bits would have to identify not just the cores, but the virtual cores that accessed a given cache line. In the directory based protocols, the sharers list would not provide the information about which virtual core has accessed each line, since the virtual cores from the same core share even the L1 cache. Therefore, the directory protocols would require the addition of bits to identify which virtual core accessed each line of the last private cache level, similarly to the A-Bits of snoop protocols. However, the TLB based mechanisms would behave the same as explained in this thesis, since the TLB already provides ways to identify to which virtual core a TLB entry belongs, otherwise it would not be possible to execute different processes inside the same multithreaded core. Hence, the TLB based mechanisms present a better support for multithreaded cores.

4 EVALUATION OF THE MECHANISMS FOR DYNAMIC DETECTION OF THE COMMUNICATION PATTERN

In this chapter, we explain how we evaluated our proposals. We execute the applications inside a simulator to detect the communication patterns. We use simulation just to allow the evaluation of our proposals, since most of them require hardware modifications. If we implemented our mechanisms on a real processor, there would be no need for simulation. To evaluate the performance, we use the communication patterns obtained in the simulator to map the threads on a real machine. We still do not migrate the threads while running the applications. Dynamic migration requires an algorithm to detect when the communication pattern changes (MA et al., 2009), as well as modifications to the scheduler of the operating system. These are beyond the scope of this work, which is to present mechanisms that dynamically detect the communication patterns.

4.1 Implementation inside the simulator

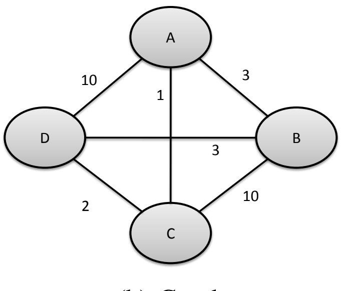
We implemented all our proposals inside the Simics simulator (MAGNUSSON et al., 2002). The cache coherence based mechanisms were evaluated with *Ultrasparc II* simulated processors. For the snoop mechanisms, we modified the original cache memory module of Simics, called *g-cache*. This module implements a set associative cache memory that uses a MESI based protocol to keep the coherence. To simulate the main memory, we used the *trans-staller* module. Simics modules present a reasonable simulation speed. The accuracy of the results are decreased because they do not consider any kind of competition between the resources, such as interconnection contention and overloaded input and output buffers. However, since the memory timings are more influent on the performance evaluations, there is no significant loss on the accuracy of the communication patterns.

For the directory based mechanisms, we modified protocols from the GEMS/Ruby project (MARTIN et al., 2005). GEMS is a set of modules for Simics that enables detailed simulation of multiprocessor systems, including Chip-Multiprocessors. There are two main modules: Opal and Ruby. Opal implements an out-of-order superscalar pipeline with several features, such as branch predictors and dynamically scheduling of instructions. Ruby is a timing simulator of a multiprocessor memory system that models caches, interconnections and memory controllers. We only used Ruby due to simulation speed constraints. Although Ruby is much more accurate than the original timing modules of Simics, we implemented only the directory based protocols in Ruby because it does not provide any snoop based protocols for multicore architectures.

The centralized directory based mechanism used the *MOESI_SMP_directory* of Ruby as basis. This protocol assumes that each node consists of a processor, private L1 and L2

	A	**B**	**C**	**D**
A	▨	3	1	10
B	3	▨	10	3
C	1	10	▨	2
D	10	3	2	▨

(a) Communication Matrix



(b) Graph

Figure 4.1: Communication Matrix and the corresponding Communication Graph.

caches, and it can be used to model a CMP with private caches. The distributed directory based mechanism used the *MSI_MOSI_CMP_directory* of Ruby as basis. This protocol assumes that each node consists of a processor with private L1 caches and one shared L2 cache. Inclusion is maintained between the L2s and the L1s, and a sharers list is kept in each L2 cache line.

The TLB based mechanisms were evaluated with *x86* based processors, because Simics does not provide the source code of the TLB of the *Ultrasparc II* processor. We performed two distinct modifications to the *x86_tlb* module: one to simulate our software-managed TLB mechanism, and one to simulate our hardware-managed TLB mechanism. We also used Simics original memory timing modules on the simulations.

4.2 Thread mapping algorithm

After the generation of the communication matrix, it is necessary to map the threads. The mapping problem is known to be NP-Hard (BOKHARI, 1981), consequently, finding the optimal solution becomes infeasible when the number of threads grows. Therefore, heuristic algorithms must be employed to determine the mapping in reasonable time, with results as similar as possible to the perfect mapping. Methods such as the Dual Recursive Bipartitioning produces good results and are available on the software Scotch (SCOTCH, 2010). However, for 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 (CRUZ et al., 2011, 2012).

This problem consists of, given a complete weighted graph $G = (V, E)$, it must be found a subset M of E in which every vertex of V is met by exactly one edge of M, and the sum of the weights of the edges of M is maximized. According to (OSIAKWAN; AKL, 1990), this problem can be solved by the Edmonds matching algorithm in polynomial time, and a parallel algorithm can solve the problem with a time complexity of $O(\frac{N^3}{P} + N^2 \lg N)$, where N is the vertex 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

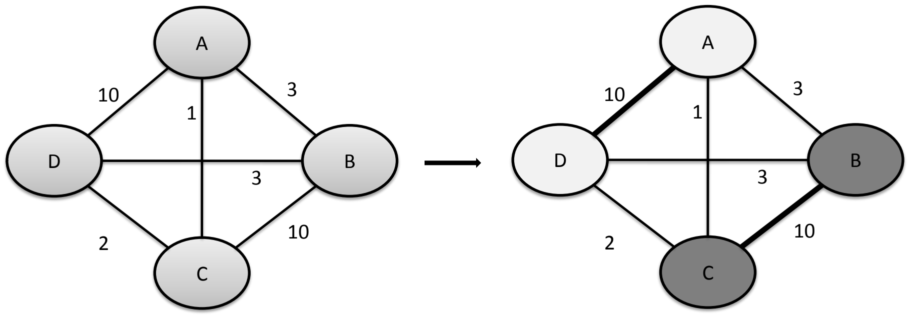


Figure 4.2: The Matching Problem.

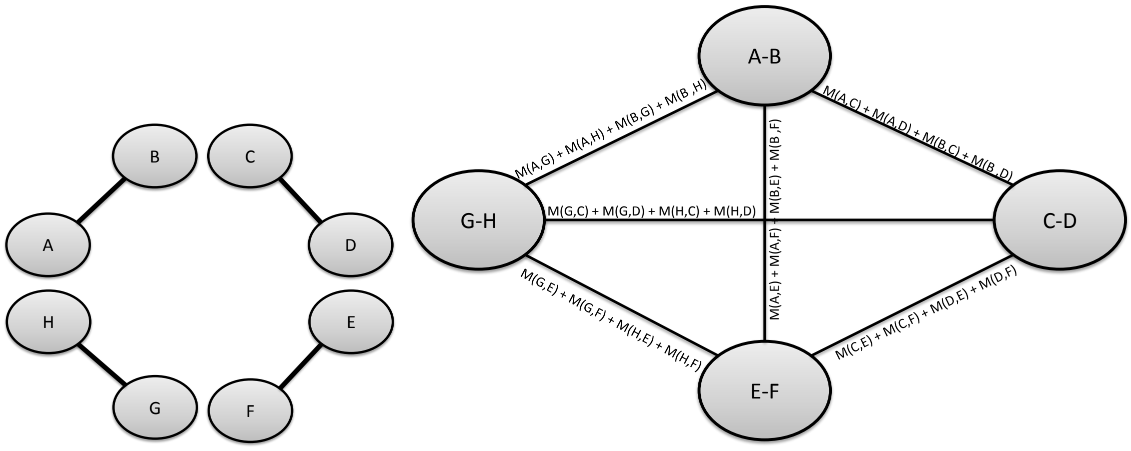
the communication matrix, as exemplified in Figure 4.1. The graph is processed by the matching algorithm, which outputs the pairs of threads so that the amount of communication is maximized. This is an extremely relevant information, since, in general, there are few cores sharing each cache. Figure 4.2 shows the result that the matching algorithm would produce for the given graph.

On many architectures, there are only 2 cores sharing each L2 cache, therefore, map threads on them with the matching algorithm is straightforward. However, there are architectures in which more than 2 cores share each 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. We generated this matrix using the following heuristic function:

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

where x, y, z and k are threads, (x, y) and (z, k) are the matchings found at the previous step, and $M_{(i,j)}$ is the amount of communication between threads i and j. The result obtained with the heuristic function is represented in Figure 4.3. Although this does not guarantee that the result will contain the pairs of pairs with the most amount of communication, since the communication matrix does not provide information about groups with more than 2 threads, it is a reasonable approximation and keeps the time and space complexity polynomial.

However, the number of cores sharing a cache may not be 2^x, where x is an integer. In this case, the matching algorithm is not able to group all the threads properly. Considering an architecture with 6 cores and 2 caches, where each cache is shared by 3 cores, for some given graph, the matching algorithm would produce the result shown in Figure 4.4(a). As can be seen, the resulting graph contains 3 disconnected graphs, but the target architecture has only 2 caches. To overcome this issue, we sort the pairs found according to the edge weight, and group only the ones that maximizes the total amount of communication, as in Figure 4.4(b). Then, a bipartite graph is generated, as exposed in Figure 4.4(c), and the matching algorithm can be applied again to group the threads for the target architecture, as show in Figure 4.4(d). The graph must be bipartite in order to prevent matchings between threads already grouped, and between threads that were not grouped yet. This procedure can be modified to map any number of threads.


B
C
A
D
H
E
G
F
A-B
G-H
C-D
E-F
M(A,G) + M(A,H) + M(B,G) + M(B ,H)
M(A,C) + M(A,D) + M(B,C) + M(B ,D)
M(G,C) + M(G,D) + M(H,C) + M(H,D)
M(A,E) + M(A,F) + M(B,E) + M(B ,F)
M(G,E) + M(G,F) + M(H,E) + M(H,F)
M(C,E) + M(C,F) + M(D,E) + M(D,F)

(a) Matching from previous step. (b) Graph obtained after applying the heuristic function.

Figure 4.3: Heuristic used to generate new communication graphs from previous matching.

4.3 Platform

We have collected the communication pattern inside Simics, while the performance evaluation was done executing the applications on the real machine.

4.3.1 Simulated environment

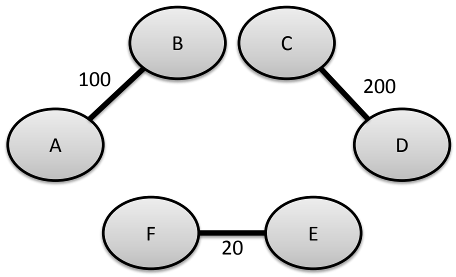
Figure 4.5 shows the memory hierarchy of the simulated environment. Tables 4.1 and 4.2 summarize the parameters used in g-cache and Ruby, respectively. As already mentioned, we used the g-cache in the snoop coherence and TLB based mechanisms, while Ruby was employed on the directory based mechanisms.

Table 4.1: Configuration of the caches using the original Simics modules.

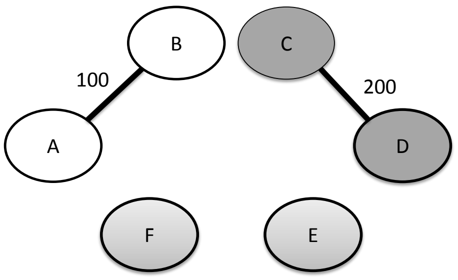
Parameter	**L1 Cache**	**L2 Cache**
Size	32KB	6MB
Number	8 Inst. + 8 Data	4 (shared by 2 cores)
Line Size	64 Bytes	64 Bytes
Set Associativity	4 Ways	8 Ways
Latency	2 Cycles	8 Cycles
Protocol	Write-through	Write-back, MESI

4.3.2 Real machine

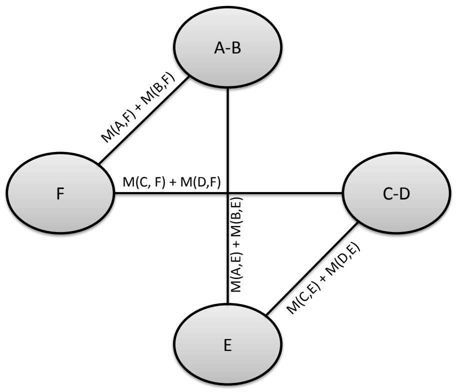
The real machine used for the performance evaluation consisted of 2 quadcore *Intel(R) Xeon(R) CPU E5405* processors. Figure 4.6 shows the memory hierarchy obtained with the Hwloc tool (BROQUEDIS et al., 2010). In this machine, there are two levels of the memory hierarchy to be exploited by thread mapping: the L2 cache and the intrachip interconnection. Cores that share the L2 cache communicate faster than cores that use the intrachip interconnection, which communicate faster than cores that are on different processors.



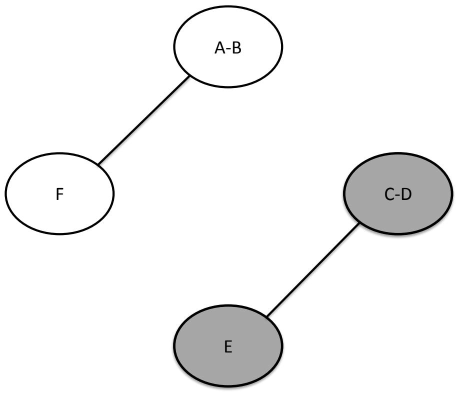
(a) Matching found.



(b) Group only the threads that maximize the comunication.



(c) Re-generate the graph.



(d) Final result.

Figure 4.4: Steps when the number of cores sharing a cache is 3.

For a better understanding of the performance results, besides the execution time, we analyzed some events on the real machine. The target architecture allows some events to be monitored by a set of hardware performance counters (INTEL, 2009). The hardware performance counters were monitored using the PAPI tool (MOORE; RALPH, 2011). The evaluated events were:

Invalidation messages – This event is represented by the counter *BUS_TRANS_INVAL*, which counts all invalidate transactions on the bus. These invalidate transactions are generated when a store operation misses the L2 cache or hits a shared line in the L2 cache.

L2 cache misses – This event is represented by the counter *L2_RQSTS.I_STATE*, which

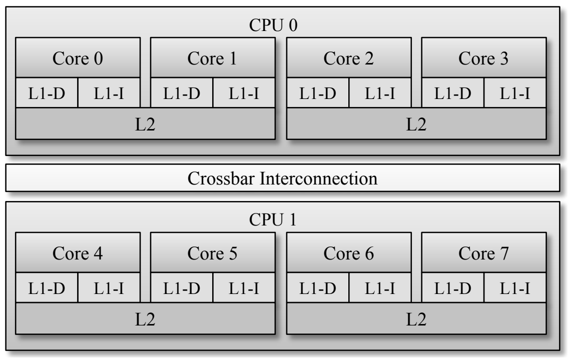


Figure 4.5: The memory hierarchy simulated in Simics.

Table 4.2: Configuration of the caches using Ruby.

Parameter	L1 Cache	L2 Cache
Size	256KB	8MB
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	MSI/MOSI	MSI/MOSI

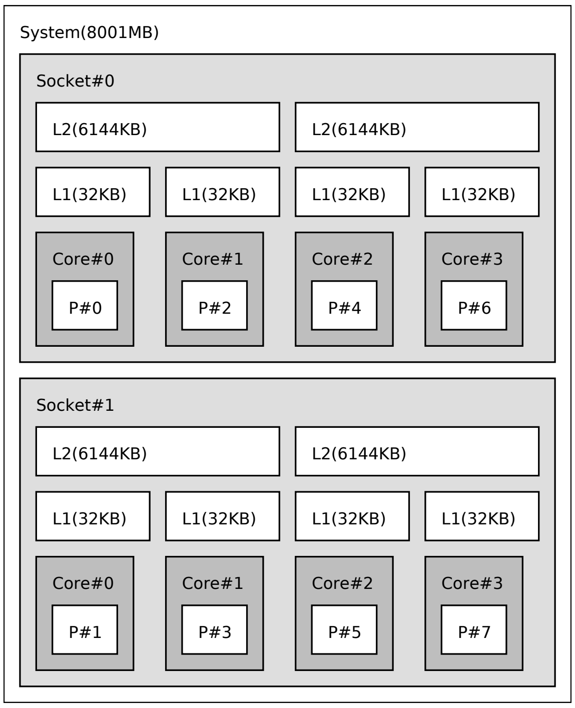


Figure 4.6: The real machine used in the performance evaluations.

counts all requests that miss the L2 cache. This includes L1 data cache reads, writes, L1 data prefetch requests, and instruction fetches.

Snoop transactions – This event is counted by the counter *BUS_HIT_DRV*, which counts the number of bus cycles in which the processor drives the HIT# pin to signal a hit snoop response. According to INTEL (2008), HIT# (Snoop Hit) convey transaction snoop operation results, and any front side bus agent may assert HIT# to indicate that it requires a snoop stall.

4.4 Validating the proposals using a microbenchmark

To validate the mechanisms, a producer-consumer application was written using the OpenMP API, with one thread being the producer, and the other the consumer. Algorithm 4.1 contains the pseudocode of the producer-consumer developed. There is a vector that is shared by the producer and consumer threads. Initially, the producer produces the data of the vector. Afterwards, the consumer consumes the vector. This process is repeated several times.

```
Algorithm:ProducerConsumer
begin Producer
    for s from 1 to NumberOfSteps do
        for i from 1 to VectorSize do
            produce Vector[i];
        end
        Warn consumer;
        Wait for consumer signal;
    end
end
begin Consumer
    for s from 1 to NumberOfSteps do
        Wait for producer signal;
        for i from 1 to VectorSize do
            consume Vector[i];
        end
        Warn producer;
    end
end
```

Algorithm 4.1: Producer Consumer.

4.4.1 Communication pattern

The communication pattern is shown at Figure 4.7. Darker cells mean more communication. We only show the results of 1 cache coherence based mechanism and 1 TLB based mechanism because the other results were very similar. The main difference between the results from the cache coherence and TLB based mechanism is that threads 0 and 1 communicate more than the other threads in the cache coherence based mechanism compared to the TLB based mechanism. This difference occurs because there are much more updates in the communication matrix of the cache coherence based mechanism than the TLB based mechanism, since invalidation messages are much more frequent than TLB misses.

4.4.2 Performance results

We have performed experiments with the producer-consumer benchmarks by running the application with the 4 possible configurations for the target architecture. The configurations are: the original operating system scheduler, threads mapped to cores that share

Table 4.3: Configuration of the caches of the real machine.

Parameter	L1 Cache	L2 Cache
Size	32KB	6MB
Number	8 Inst. + 8 Data	4 (shared by 2 cores)
Line Size	64 Bytes	64 Bytes
Set Associativity	8 Ways	24 Ways

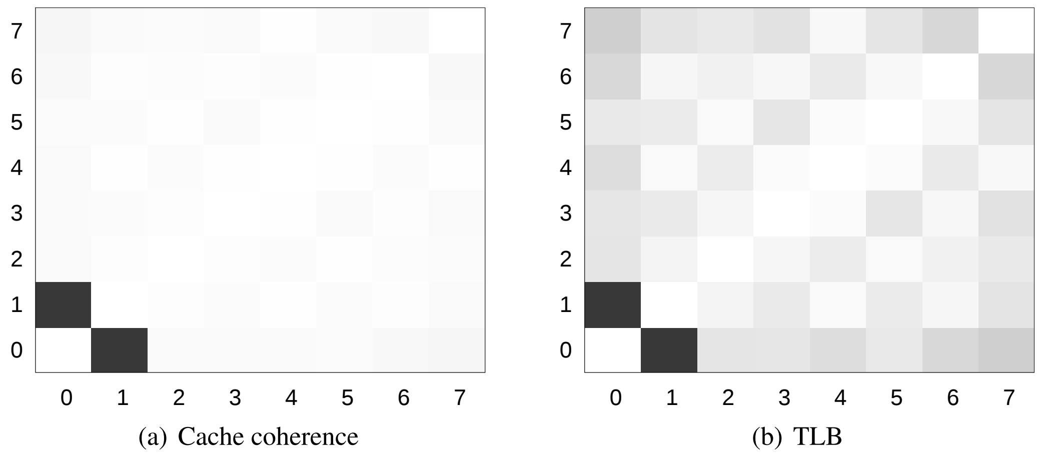


Figure 4.7: Communication patterns of the producer-consumer benchmark.

the L2 cache, threads mapped to cores that share the same processor, and threads mapped to cores located on different processors. We configured the size of the vector to fit into the L2 cache to minimize the noisy from evictions due to line replacement. Figure 4.8 shows the results.

As expected, the results are better when the threads are mapped to cores which are nearer in the memory hierarchy. When the threads are mapped to cores that share the L2 cache, the number of invalidation messages and snoop transactions are null, which is correct because the cache lines corresponding to the producer consumer are considered private by the cache coherence protocol. Also, as there is no invalidation message to evict the cache line from other caches, the vector is almost always found on the cache, dropping the caches misses to almost zero.

4.5 Workload

In this section, we present the NAS Parallel Benchmark (NPB) workload, used to evaluate our proposals. The NPB has its applications derived from computational fluid dynamics (CFD) codes and it is composed by applications and kernels (JIN; FRUMKIN; YAN, 1999). NPB applications and kernels perform representative computation and data communication of CFD codes. These characteristics allow us to evaluate the impact of thread mapping on multi-threaded programs over multi-core machines. NPB has several standard inputs, and the one used for the evaluation is the W input size, because it is the most recommended for simulation. We focus on describing the communication patterns of the applications, since they are the most relevant information for our work and are used to evaluate the accuracy of our proposals.

4.5.1 Tools used to obtain the communication pattern baseline

Regarding the producer-consumer benchmark, it is easy to verify if the communication patterns obtained with our proposed mechanism are correct, because the behavior of the application is straightforward. However, in more complex applications, it may be difficult to discover an accurate communication pattern by looking at the source code. Hence, to detect the most accurate communication pattern possible to serve as baseline, we can use an old fashioned static mechanism. In the technical report CRUZ; NAVAUX

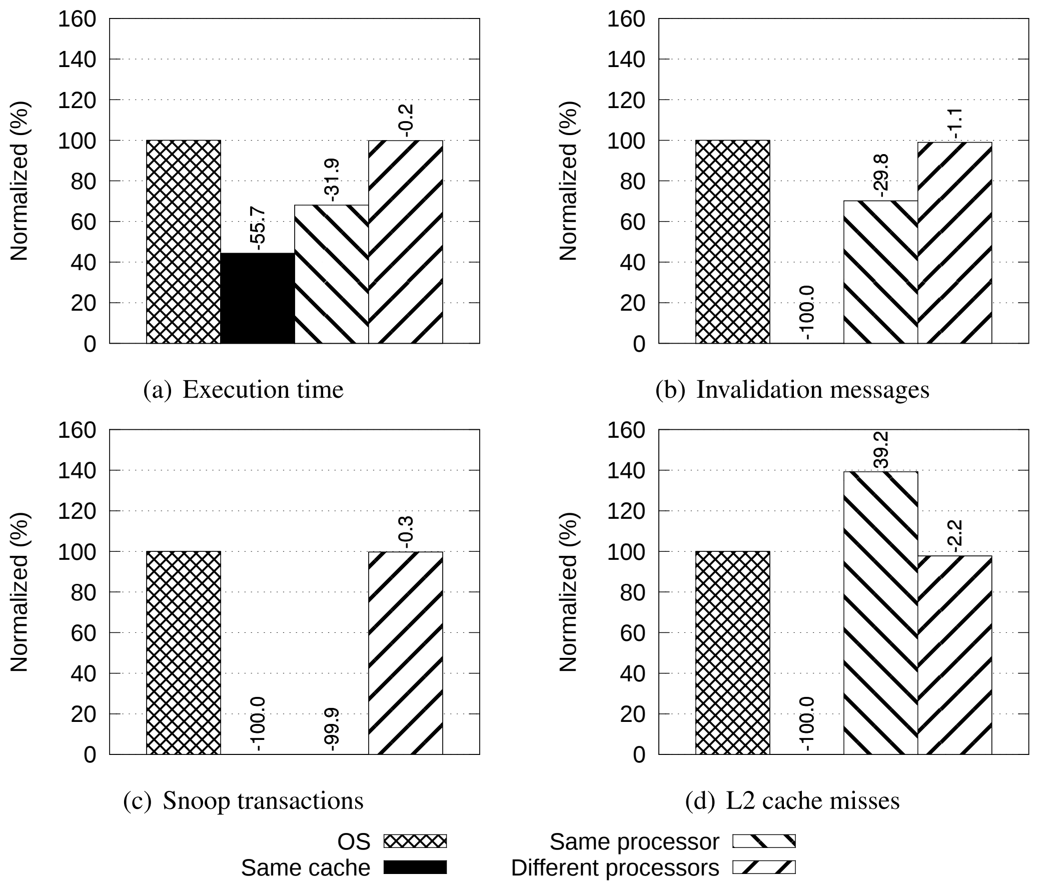


Figure 4.8: Performance results with the producer-consumer benchmark.

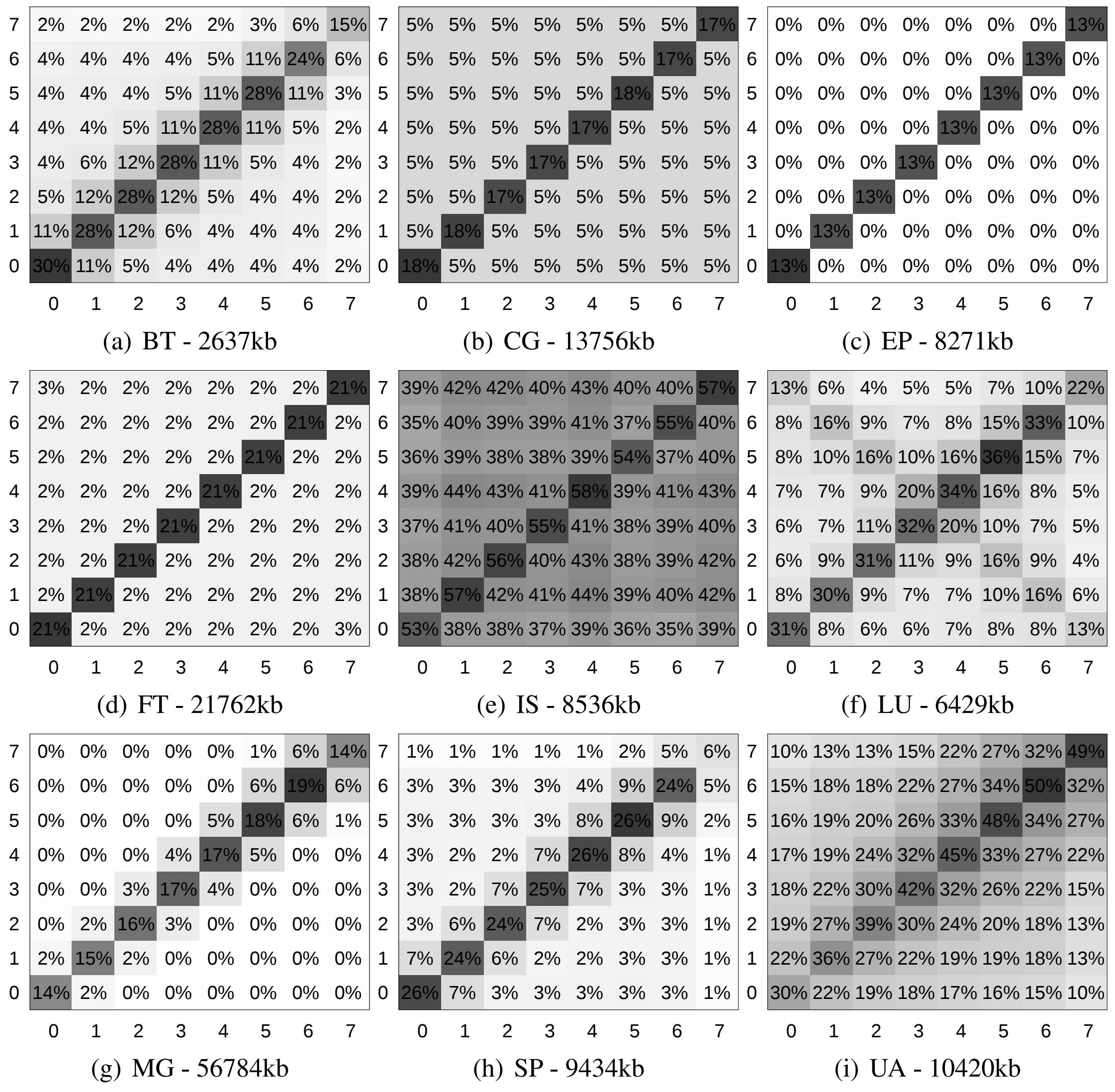
(2010), some tools to monitor the memory access are evaluated. Table 4.4 summarizes the results.

We chose Simics with the C Module (CRUZ et al., 2011, 2012) to detect the communication pattern baseline because it presents a good trade-off between performance and accuracy. Simics was instrumented (MAGNUSSON et al., 2002) to register memory access information such as the moment when the access happened, the identifier of the thread that generated the access, the memory address, the operation type (read, write or instruction fetch) and its size. However, it is necessary to filter the accesses to be registered, so that only the memory accesses performed by the evaluated application are stored in the trace file. Although Simics API implements tools to determine which task is running in each processor, it becomes unstable when the number of processors simulated is high. To overcome this issue, the Linux kernel inside Simics was modified to warn the simulator about which task was being scheduled to run. This way, the memory trace module is able to detect if a memory access was performed by the application being analyzed.

The memory traces alone are not enough to guide the thread mapping. The traces must be analyzed to discover the communication pattern. For this purpose, a tool was developed in the C++ language to read the trace files and generate the statistics. The tool evaluates how much memory each thread uses, both the total and shared memory, how many access were performed, among other statistics.

Table 4.4: Comparative between tools that can be used to detect the memory accesses.

Tool	Type	Performance	Simulates SMP?	Accuracy
Qemu	Emulator	High	No	Low
Bochs	Emulator	Medium	Yes	Medium
Simics (Python)	Simulator	Low	Yes	High
Simics (C Module)	Simulator	Medium	Yes	High
Valgrind	DBA	High	No	Low
Pin	DBA	High	Yes	Medium



(a) BT - 2637kb (b) CG - 13756kb (c) EP - 8271kb

(d) FT - 21762kb (e) IS - 8536kb (f) LU - 6429kb

(g) MG - 56784kb (h) SP - 9434kb (i) UA - 10420kb

Figure 4.9: Amount of memory used by the applications of NPB.

4.5.2 Communication pattern baseline

For the communication pattern baseline, two metrics were separately considered to evaluate the communication: the amount of memory shared by the threads and the number of accesses performed to a block of memory that is shared. The amount of memory shared by the threads metric is more suitable to applications in which the number of accesses to the shared memory is insignificant compared to the number of accesses to the private

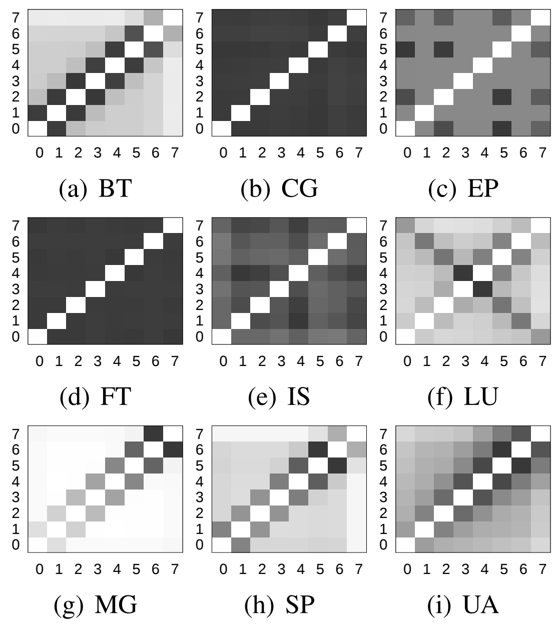


Figure 4.10: Amount of memory shared by the threads of NPB.

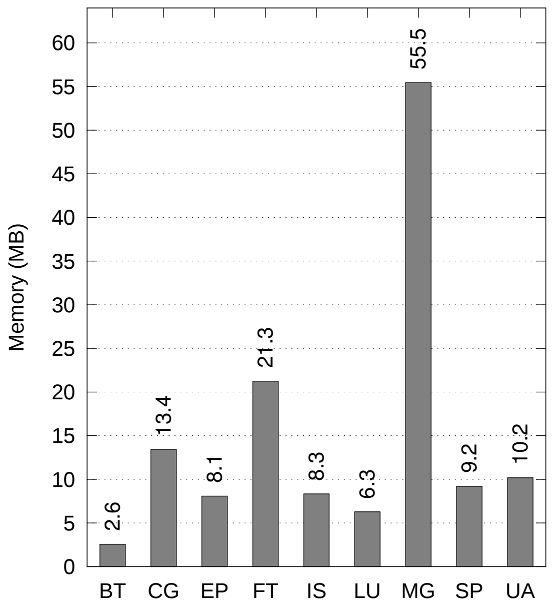


Figure 4.11: Amount of memory used by the applications of NPB.

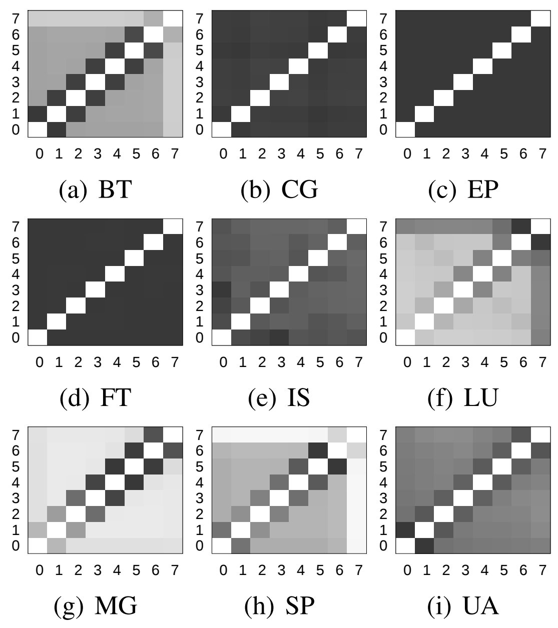


Figure 4.12: Number of accesses to the shared memory (5 million cycles time window)

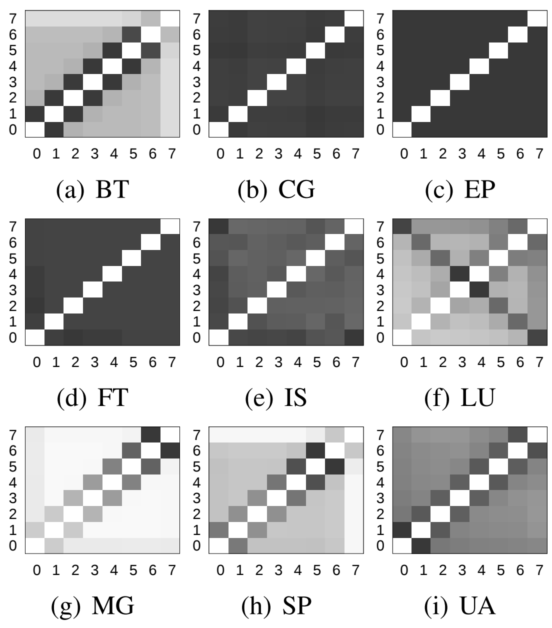


Figure 4.13: Number of accesses to the shared memory (50 million cycles time window)

memory. On the other hand, the amount of accesses to the shared memory metric are better to describe the behavior of applications that present a huge amount of accesses to the shared memory.

Figure 4.11 shows the amount of memory used by the benchmarks. Figure 4.9 and 4.10 show the amount of memory shared between the threads. Each cell *(i, j)* represents the amount of memory shared between threads i and j. When i equals j, it represents the amount of memory accessed by thread i. Darker cells represent more memory. We omit the diagonal in some Figures to enhance the differences between the cells corresponding to pairs of different threads, thereby the darkest cell does not correspond to the total amount of memory. Figures 4.12 and 4.13 show the amount of accesses performed to a block of shared memory considering a 5 million and 50 million cycles time window, respectively. Each cell *(i, j)* represents the number of access to a block of memory shared between threads i and j. Darker cells represent more access to the shared memory.

BT is an application that presents most of its communication between neighboring threads. This is very common when the application is based on domain decomposition, where most of the communication happens between neighbors and most of the shared data is located on the borders of each sub-domain. The domain decomposition pattern is also present in MG, SP and UA. These 4 applications do not show any significant difference comparing the matrix of the amount of memory to the matrices of number of accesses to the shared memory, for both 5 and 50 million cycles time window.

The application LU also presents the domain decomposition pattern, which is more evident with a 5 million cycles time window. However, when we increase the time window to 50 million cycles, it also presents huge amount of communication between the most distant threads. Regarding the amount of shared memory, the communication between the most distant threads is also clearly identified. IS also presents significant differences between the matrices. The amount of memory matrix does not define a clear pattern. However, the number of accesses to the shared memory matrices show traces of a domain decomposition pattern, especially when we consider a 5 million cycles time window.

The applications CG, EP and FT present homogeneous communication patterns. Homogeneous means that their communication patterns are expected to present approximately the same amount of communication among the threads. One key difference from EP to CG and FT is that the threads of EP almost do not share any memory. This difference can be seen in Figure 4.9, in which CG and FT share about 5% and 2% of memory, respectively, while EP shares almost only 0%. On the other hand, heterogeneous pattern means that the amount of communication among the threads varies, as in BT, LU, MG, SP and UA. It is expected that applications with heterogeneous communication patterns to benefit more from thread mapping than applications with homogeneous communication patterns.

5 RESULTS

In this chapter, we present the results we obtained by using our proposed mechanisms on the benchmarks. First, we show the communication patterns we discovered and compare them to the baseline, described in Section 4.5.2. Afterwards, we detail the performance improvements we achieved by mapping the threads using the communication patterns. For the snoop based mechanisms, we show only the results from the snoop for shared caches. Regarding the directory based mechanisms, we show only the distributed directory based mechanism. We omit the other results because the results from the snoop private caches were very similar to the snoop shared ones, and the results from the centralized directory were similar to the ones from the distributed directory. From now on, we will call the approach for the snoop cache coherence *SNOOP*, the directory cache coherence *DIRECTORY*, the software-managed TLB *TLB-SM*, and the the approach for hardware-managed TLB *TLB-HM*.

5.1 Communication Patterns

Figures 5.1 and 5.2 show the communication patterns of the NPB applications for the TLB-SM and TLB-HM, respectively. The size of the TLBs was 64 entries for both approaches, with a 4-way set associative. This is the default size of the TLB in UltraSparc, as well as the size of the Level 1 TLB in the Nehalem architecture. For the TLB-SM, only in 1% of the TLB misses a search for matches was performed. We also simulated the TLB-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. TLB-HM was evaluated with 10 million cycles between each search for matches.

Figure 5.3 shows the communication patterns for the SNOOP. The DIRECTORY is very sensitive to the cache size, therefore, we evaluate the DIRECTORY with two different cache sizes to investigate this behavior. Figures 5.4 and 5.5 show the communication patterns for DIRECTORY with 32kb and 256kb cache sizes, respectively. We used a 32kb cache size for DIRECTORY to evaluate the mechanism in architectures in which only the L1 cache is private, such as in *Harpertown* (INTEL, 2008). We used a 256kb cache size for DIRECTORY to evaluate the mechanism in architectures in which the L2 cache is private, such as in *Nehalem* (INTEL, 2011). We also simulated the DIRECTORY with 512kb cache size, which is present in architectures such as the AMD 6200 Processors (AMD, 2011), but we chose not to show the communication patterns because only the SP application behaved significantly different compared to a 256kb cache size.

The communication patterns were used by our thread mapping algorithm, explained in Section 4.2, to map the threads on cores according to the amount of communication. Table 5.1 shows the thread mapping, in which each cell contains the id of the thread.

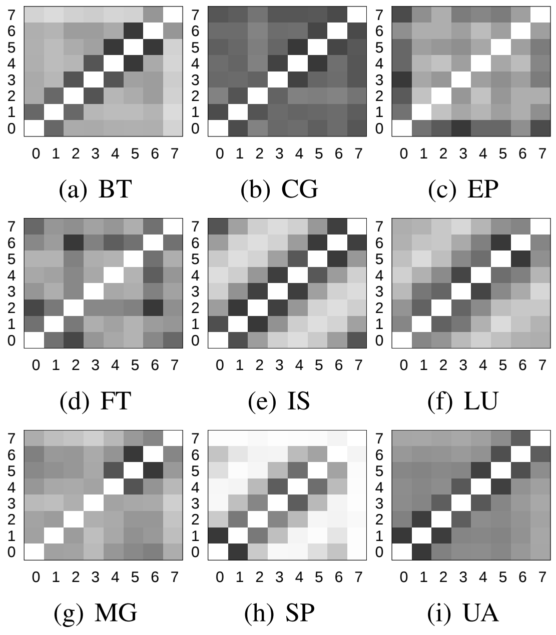


Figure 5.1: Communication patterns of the applications from NPB discovered with the software-managed TLB mechanism.

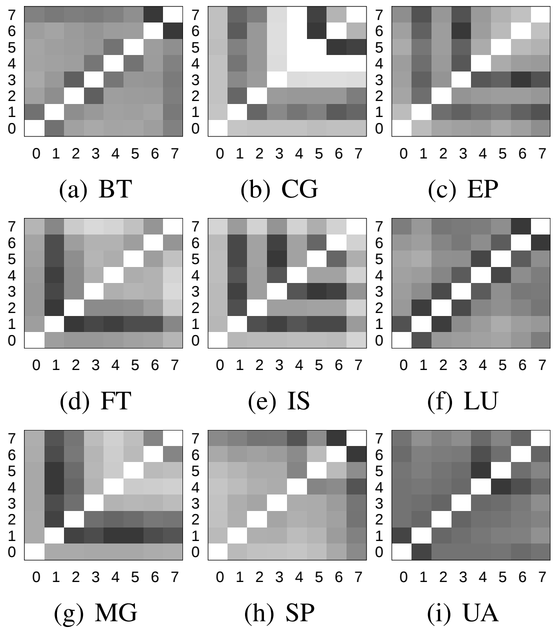


Figure 5.2: Communication patterns of the applications from NPB discovered with the hardware-managed TLB mechanism.

BT is an application that presents a domain decomposition communication pattern, where most of the shared data is located on the borders of each sub-domain and hence the communication is more evident between neighboring threads. For BT, the TLB-SM, TLB-HM, SNOOP and DIRECTORY with 256kb mechanisms were able to detect the communication pattern. TLB-HM detected more communication between thread 7 and all other threads than the other mechanisms. SNOOP detected less communication between threads 6 and 7. Furthermore, the communication between neighbors is more expressive in SNOOP. As a result, the mapping algorithm found an optimal mapping just for TLB-SM, DIRECTORY with 256kb cache size and SNOOP, while, for TLB-HM, a slightly worse mapping was found. The DIRECTORY with 32kb cache size was not able to detect the communication pattern, which degrades the quality of the mapping.

SNOOP and DIRECTORY with both cache sizes, successfully detected the communication pattern of MG, resulting in optimal mappings. TLB-SM 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 are still differences. It is important to mention that, when we simulated TLB-SM taking into account all the TLB misses, the generated pattern clearly identified the communication pattern of MG. TLB-SM, SNOOP and DIRECTORY with both cache sizes successfully detected the communication pattern of IS, however, the mapping algorithm found the optimal mapping only for DIRECTORY with 256kb cache size. It is important to note that the communication pattern baseline of IS only clearly behaved as domain decomposition when considering a 5 million cycles time window (Figure 4.12(e)).

In IS and MG, TLB-HM detected a large amount of communication between two of the threads and all the other ones. The reason for this result is the runtime behavior of the applications, which can present a challenge to TLB-HM. For instance, consider the case that the sampling is made when the threads 0 and 1 are accessing their shared

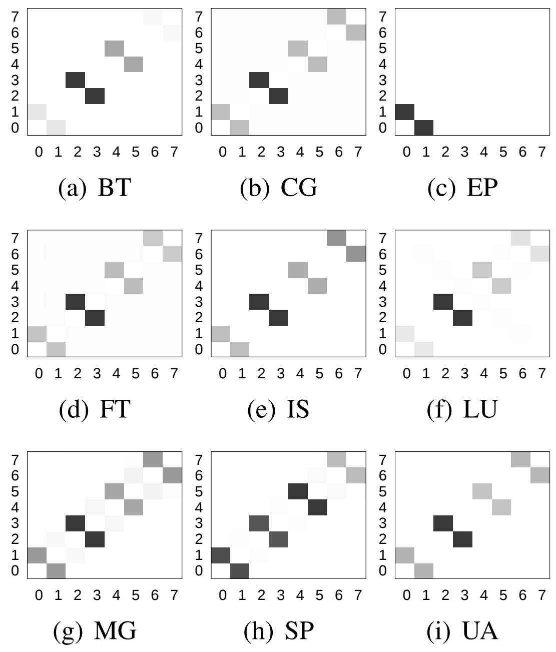


Figure 5.3: Communication patterns of the applications from NPB discovered with the snoop coherence mechanism.

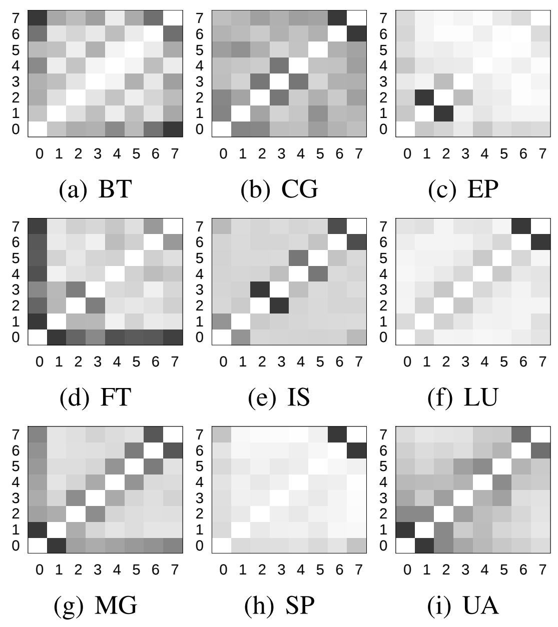


Figure 5.4: Communication patterns of the applications from NPB discovered with the directory coherence mechanism (32kb L1 cache).

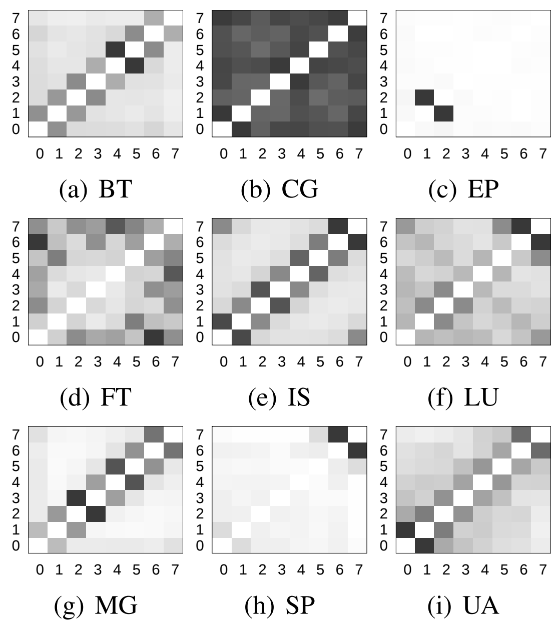


Figure 5.5: Communication patterns of the applications from NPB discovered with the directory coherence mechanism (256kb L1 cache).

data, but at the same time the other threads are working on their private data. As this situation describes a temporary behavior, it may not characterize the global behavior of the application. If this happens several times, TLB-HM would 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 among themselves; it means that the sampling was performed at an inconvenient time.

Table 5.1: Mappings obtained by applying the thread mapping algorithm.

App.	Mechanism	Processor				Processor			
		L2 Cache		L2 Cache		L2 Cache		L2 Cache	
BT	TLB-SM	2	3	0	1	4	5	6	7
	TLB-HM	6	7	0	1	2	3	4	5
	DIRECTORY-32kb	1	7	3	5	0	6	2	4
	DIRECTORY-256kb	2	3	0	1	4	5	6	7
	SNOOP	2	3	0	1	4	5	6	7
CG	TLB-SM	5	6	0	7	3	4	1	2
	TLB-HM	5	6	2	7	1	4	0	3
	DIRECTORY-32kb	0	2	1	5	6	7	3	4
	DIRECTORY-256kb	0	1	5	7	3	4	2	6
	SNOOP	2	3	4	5	6	7	0	1
EP	TLB-SM	0	3	2	5	4	7	1	6
	TLB-HM	1	7	2	5	3	6	0	4
	DIRECTORY-32kb	1	2	3	5	0	4	6	7
	DIRECTORY-256kb	1	2	0	3	5	7	4	6
	SNOOP	0	1	2	3	4	5	6	7
FT	TLB-SM	4	6	2	5	0	1	3	7
	TLB-HM	1	4	2	5	6	7	0	3
	DIRECTORY-32kb	0	1	2	3	6	7	4	5
	DIRECTORY-256kb	0	2	3	6	4	7	1	5
	SNOOP	2	3	4	5	0	1	6	7
IS	TLB-SM	1	2	3	4	5	6	0	7
	TLB-HM	3	4	5	6	1	2	0	7
	DIRECTORY-32kb	6	7	0	1	2	3	4	5
	DIRECTORY-256kb	0	1	2	3	6	7	4	5
	SNOOP	6	7	0	1	2	3	4	5
LU	TLB-SM	3	4	1	2	5	6	0	7
	TLB-HM	6	7	4	5	2	3	0	1
	DIRECTORY-32kb	6	7	4	5	2	3	0	1
	DIRECTORY-256kb	1	2	0	3	6	7	4	5
	SNOOP	2	3	4	5	6	7	0	1
MG	TLB-SM	0	1	2	3	6	7	4	5
	TLB-HM	1	5	6	7	2	4	0	3
	DIRECTORY-32kb	0	1	2	3	6	7	4	5
	DIRECTORY-256kb	0	1	2	3	6	7	4	5
	SNOOP	0	1	2	3	6	7	4	5
SP	TLB-SM	0	1	2	3	4	5	6	7
	TLB-HM	6	7	0	1	4	5	2	3
	DIRECTORY-32kb	6	7	0	1	2	4	3	5
	DIRECTORY-256kb	6	7	2	5	0	1	3	4
	SNOOP	0	1	2	3	4	5	6	7
UA	TLB-SM	0	1	2	3	4	5	6	7
	TLB-HM	0	1	2	3	4	5	6	7
	DIRECTORY-32kb	0	1	2	3	4	5	6	7
	DIRECTORY-256kb	0	1	2	3	4	5	6	7
	SNOOP	0	1	2	3	4	5	6	7

In LU, all the mechanisms detected the domain decomposition pattern. TLB-SM detected that there is also a considerable amount of communication between distant threads. Only the DIRECTORY approach, with a 256kb cache size, clearly detected the communication between the most distant threads. In the communication pattern baseline we can note that the communication between the most distant threads was detected only when the time window was increased to 50 million cycles. Therefore, this difference between the behavior of the DIRECTORY with 32kb and 256kb cache sizes is consistent with the communication pattern baseline, since an increase of the size of the cache represent the increase of the time window. The generated mappings differ a little among themselves, except the ones from TLB-HM and DIRECTORY with a 32kb cache size.

For SP, only the TLB-SM and SNOOP mechanisms were able to accurately identify the communication pattern and obtain optimal mappings. The DIRECTORY correctly detected the communication between thread pairs 0-1 and 6-7, but failed for the other threads. TLB-HM also failed, however, the results obtained by TLB-HM with SP are much better than the results with IS and MG, as none of the threads presented a large amount of communication to all others. Due to these reasons, for DIRECTORY and TLB-HM, a sub-optimal mapping was found. It is important to mention that, considering a 512kb cache size, the DIRECTORY mechanism accurately identified the communication pattern of SP.

In UA, all mechanisms except TLB-HM clearly exhibit the domain decomposition pattern. As expected, the DIRECTORY present a more accurate pattern with higher cache sizes. As in SP, the results obtained by TLB-HM with 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, even though the domain decomposition pattern obtained with TLB-HM is not as evident as with the other mechanisms, our mapping algorithm was able to find the optimal mapping. Therefore, all mechanisms resulted in optimal mappings.

SNOOP detected a strong domain decomposition pattern in both CG and FT. CG, with the TLB-SM mechanism, show traces of a domain decomposition pattern. Nevertheless, with TLB-SM, 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. With the other mechanisms, a homogeneous pattern was detected for CG. In FT, all mechanisms except SNOOP presented homogeneous patterns. EP is the application that almost does not communicate; hence all the mechanisms presented a homogeneous pattern for EP.

Regarding the TLB mechanisms, in general, the communication pattern detected by TLB-SM is more accurate. The reason is that the pattern discovery mechanism with TLB-SM is able to access more samples than TLB-HM, as all the TLB misses are handled by the operating system. TLB-HM, as explained before, suffers from unpredictable runtime behavior and bad samples. This is reinforced when analyzing CG, EP and FT, as they presented more homogeneous pattern with TLB-SM than TLB-HM. Regarding the cache coherence mechanisms, we can note that SNOOP detected stronger domain decomposition patterns than DIRECTORY. This difference is caused because the A-Bits are manipulated differently in SNOOP and DIRECTORY (sharers list). Additionally, as expected, the DIRECTORY is more accurate with higher cache sizes.

5.2 Evaluating the Performance using Thread Mapping

The results presented in this section are average values obtained by executing each benchmark 100 times. For the DIRECTORY mechanism, we used the communication patterns obtained considering a 256kb cache size. Figures 5.6, 5.7, 5.9 and 5.8 presents the execution time, number of invalidations, snoop transactions and L2 cache misses, respectively. We focused only on the L2 cache misses because the L1 caches are private and do not benefit from mapping. The results of these graphics are normalized to the time of the original scheduler of the operating system, denoted by OS. Table 5.2 contains the absolute values for number of invalidations, snoop transactions and cache misses, all divided by the execution time. Table 5.3 shows the standard deviations.

The number of invalidations, snoop transactions and cache misses in BT were greatly

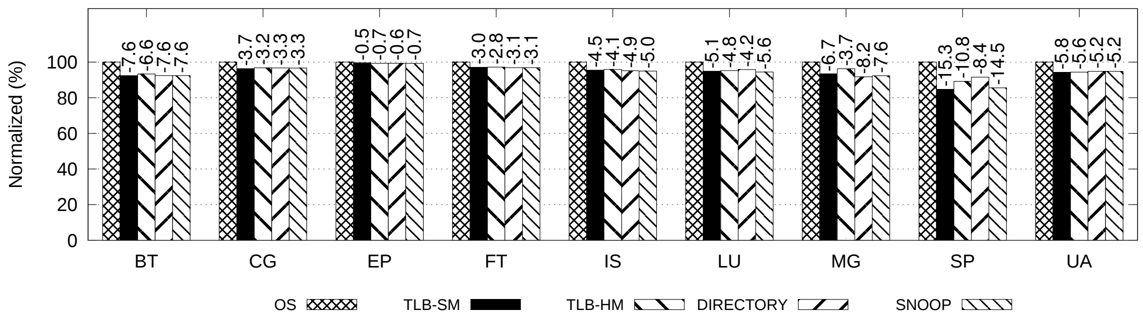


Figure 5.6: Execution time of the applications.

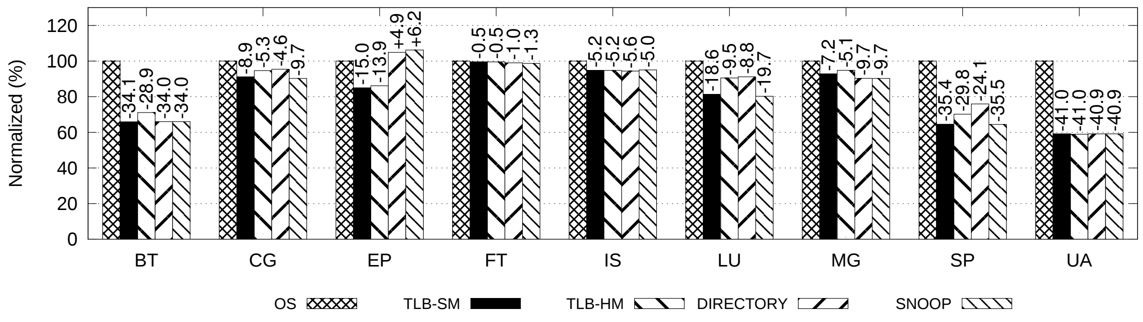


Figure 5.7: Invalidations due to the cache coherence protocol.

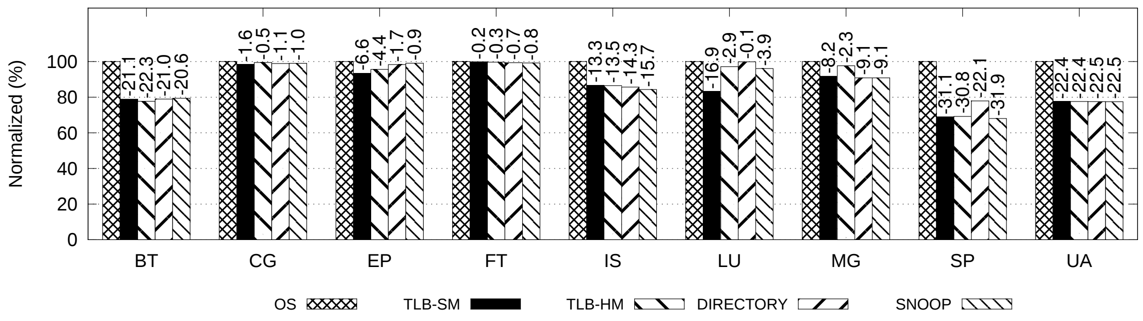


Figure 5.8: L2 cache misses.

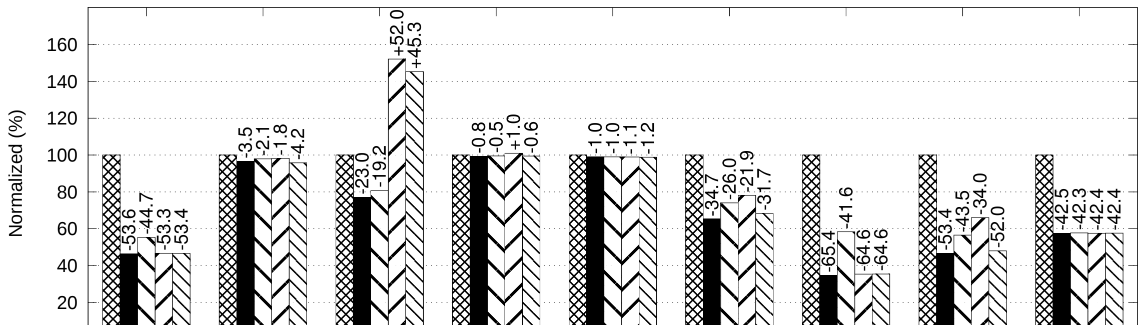


Figure 5.9: Snoop transactions.

reduced. The execution time also was reduced, but by a small factor. Except for cache misses, the results with TLB-SM, DIRECTORY and SNOOP are a little better than TLB-HM, as the mapping found for them were better. TLB-HM obtained slightly less cache misses, but the difference is small and presented high standard deviation, which makes the difference on cache misses negligible. 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 lines prefetches, 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 in 15.3% and 31.9%, respectively. TLB-SM and SNOOP performed better than TLB-HM and DIRECTORY in execution time, number of invalidations and snoop transactions, as only TLB-SM and SNOOP detected the correct communication pattern. For cache misses, the results were similar for all mechanisms, since we obtained high standard deviations. UA achieved the best reduction of the number of invalidations (41%). The improvements in UA were virtually the same for all measures, since the optimal thread mapping was found for all the mechanisms.

In LU, the performance of all mechanisms was almost the same. However, as the mappings differ a little among themselves, there is a significant difference in the reduction of the amount of invalidations and snoop transactions. Although the number of cache misses was also reduced, the standard deviation is high. It is important to notice 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%). TLB-SM, DIRECTORY and SNOOP performed better than TLB-HM, since 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 mentioned in the last sentence, MG was the one that presented the lowest reduction of the number of invalidations. This is the reason that MG, despite having the best relative reduction of snoop transactions, presented lower reductions of cache misses.

As already stated, thread mapping improves performance by mapping threads that communicate 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 TLB-SM and SNOOP was able to slightly reduce the number of invalidations compared to the others. For the other measures of CG, the standard deviations make the small improvements negligible.

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 alter drastically the normalized

Table 5.2: Execution time and number of invalidations, snoop transactions and L2 cache misses per second.

Parameter	Map.	BT	CG	EP	FT	IS	LU	MG	SP	UA
Execution Time (seconds)	OS	0.74	0.13	0.48	0.1	0.06	2.39	0.23	2.53	2.19
	TLB-SM	0.68	0.13	0.47	0.1	0.06	2.27	0.22	2.14	2.06
	TLB-HM	0.69	0.13	0.47	0.1	0.06	2.27	0.22	2.25	2.06
	DIREC.	0.68	0.13	0.47	0.1	0.06	2.29	0.21	2.31	2.07
	SNOOP	0.68	0.13	0.47	0.1	0.06	2.25	0.21	2.16	2.07
Invalidations / second	OS	9845216	3831746	121230	16154353	9754232	14457991	35970058	17749230	7361187
	TLB-SM	7019908	3624698	103558	16571898	9681120	12395757	35792412	13535357	4609197
	TLB-HM	7499308	3747079	105117	16544292	9637287	13745080	35439765	13956912	4600673
	DIREC.	7034617	3781271	128025	16510459	9676625	13757114	35392434	14719406	4591157
	SNOOP	7035756	3578149	129663	16454333	9758512	12290195	35149772	13374610	4590046
Snoop Transac-tions / second	OS	7196937	10374266	27870	5172957	11461581	12706165	4093348	10668132	5008487
	TLB-SM	3612138	10395271	21560	5288628	11889910	8739948	1519446	5874685	3055559
	TLB-HM	4263300	10492865	22666	5298599	11830896	9881274	2482490	6757793	3064284
	DIREC.	3634426	10532198	42647	5392707	11918312	10357332	1579034	7695265	3041511
	SNOOP	3632095	10277555	40774	5310674	11927062	9187466	1570062	5989416	3042851
L2 Misses / second	OS	248962	1144400	3365	460250	1007312	656734	939658	339850	741887
	TLB-SM	212403	1169066	3159	473133	914644	575242	924153	276327	610845
	TLB-HM	207314	1176111	3240	472221	908205	669864	953271	263512	610188
	DIREC.	212748	1170316	3329	471963	907353	684764	930349	289115	606718
	SNOOP	213956	1171742	3358	471289	894176	668051	924200	270403	606226

Table 5.3: Standard deviations for the performance experiments.

Parameter	Map.	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%
	TLB-SM	4.15%	2.68%	1.98%	6.83%	4.62%	0.2%	2.82%	0.11%	0.25%
	TLB-HM	0.79%	4.62%	1.87%	6.13%	11.11%	1.17%	3.11%	0.11%	1.21%
	DIREC.	0.41%	1.46%	0.79%	2.08%	2.02%	0.21%	1.03%	0.99%	1.35%
	SNOOP	0.45%	1.41%	0.9%	2.02%	2.86%	1.07%	5.74%	5.08%	1.21%
Invalidations	OS	4.68%	1.45%	30.68%	0.88%	1.52%	4.55%	1.64%	4.75%	1.92%
	TLB-SM	3.41%	0.92%	22.79%	0.58%	0.68%	0.16%	2.22%	0.42%	0.97%
	TLB-HM	5.69%	1.37%	18.89%	0.48%	0.86%	1.29%	1.95%	8.36%	1.3%
	DIREC.	3.52%	1.64%	24.56%	1.38%	1.14%	0.46%	1.75%	3.32%	0.76%
	SNOOP	3.52%	1.64%	23.14%	0.95%	1.51%	3.04%	2%	0.72%	0.75%
Snoop Transac-tions	OS	5.08%	1%	32.53%	1.02%	0.78%	8.45%	7.75%	8.35%	5.79%
	TLB-SM	5.72%	0.47%	52.32%	0.73%	0.81%	1.21%	12.03%	1.29%	3.56%
	TLB-HM	6.34%	1.13%	44.21%	1.4%	1.01%	2.6%	1.03%	4.6%	3.36%
	DIREC.	5.28%	0.58%	64.81%	1.32%	0.95%	1.74%	11.1%	5.28%	4.09%
	SNOOP	5.25%	1.45%	66.8%	1.69%	0.99%	0.89%	11.15%	2.36%	4.06%
L2 Misses	OS	25.74%	1.92%	41.1%	5.28%	2.75%	11.32%	4.6%	30.04%	8%
	TLB-SM	23.89%	2.37%	38.4%	5.18%	3.3%	26.41%	4.96%	36.94%	15.03%
	TLB-HM	22.82%	2.98%	32.14%	5.25%	3.3%	14.94%	7.47%	37.48%	15.12%
	DIREC.	23.97%	2.52%	29.69%	5.08%	3.92%	13.07%	4.72%	43.77%	15.06%
	SNOOP	23.83%	1.27%	29.15%	5.31%	3.56%	15.28%	4.73%	37.01%	15.04%

results, as depicted in Figures 5.7 and 5.9. Additionally, by looking at the standard deviations of EP, we see that they surpass the improvements and losses, showing that no improvements can be achieved by thread mapping in EP, which is the expected result.

Regarding IS, the communication pattern 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.

Table 5.4: 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%

5.3 Overhead of the Mechanisms

In this section, we calculate the overhead of our proposed mechanisms.

5.3.1 Cache coherence based mechanisms

There is no time overhead for the cache coherence based mechanisms. The space overhead imposed by the snoop based mechanism for shared caches consists of two A-Bits for every L2 cache line and one sub-matrix per L2 cache. The A-Bits for each cache require 24576 bytes. Each sub-matrix has 2 columns and 8 rows with 32 bit cells, requiring 64 bytes. As there are 4 L2 caches, the total space required is 98560 bytes. There is a total of 24mb of cache memory, hence the size overhead is less than 0.4%. The L1 cache requires no modification because it is configured with a write-through policy.

Regarding the snoop private and the distributed directory based mechanisms, the space overhead is one sub-matrix for each L2 cache. Each sub-matrix has 2 columns and 8 rows with 32 bit cells, requiring 64 bytes. As there are 4 L2 caches, the total space required is only 256 bytes. For the centralized directory, the space overhead is the communication matrix attached to the directory. It contains 8 rows and 8 columns with 32 bits cells, requiring a total of only 256 bytes. Therefore, snoop private and both directory based mechanisms present a negligible space overhead.

5.3.2 TLB based mechanisms

To decrease the overhead of our proposed mechanism, we used set associate software-managed and hardware-managed TLBs. As stated in Section 3.2, this reduces the time complexity of the algorithms that detect the communication. The associativity used in our experiments was 4. We do not present the space overhead, because it is not relevant for the TLB based mechanisms, as explained in Section 3.3.

Table 5.4 presents statistics for TLB-SM. It shows the TLB miss rate, the percentage of TLB misses for which we run TLB-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. Additionally, less than 1% of total TLB misses are considered for sampling, as we are only interested in TLB misses due to data accesses. The reason that we monitor only the data accesses is explained in Section 2.

We measured the amount of cycles spent on the communication discovery routine of TLB-SM, and the value obtained was only 231 cycles. Due to this low value, all applications except IS presented a very low overhead of less than 1% for TLB-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%).

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 TLB-SM, which depends on the number of TLB misses the application causes. The communication discovery routine for TLB-HM requires 84297 cycles to execute. This large difference between the TLB-SM and TLB-HM cycles can be explained by the time complexity that we calculated in Section 3.2. As we performed the sampling only every 10 million cycles, the overhead of TLB-HM is less than 0.85% for our experiments.

6 CONCLUSION AND FUTURE WORK

Parallel architectures have become the standard solution to keep the performance of large scale architectures increasing. One of the main concerns in these architectures is the communication between the threads, since it implies in data movement among the cores, leading to performance loss and energy consumption. Additionally, it is expected that the upcoming increase on the number of cores will aggravate the problem. Therefore, it is important to research and develop mechanisms to optimize the communication. One of the mechanisms is called thread mapping, which allows a better usage of the resources by mapping the threads to specific cores according to some policy. By using the communication between the threads as policy, the usage of the memory hierarchy is optimized.

The analysis of the state-of-art of thread mapping techniques show that most of the researches focus on mapping the process of applications based on message passing. These techniques are straightforward comparing to mapping the threads of applications based on shared memory, because the communication with message passing is explicit, while with shared memory it is implicit. Furthermore, current thread mapping mechanisms can be categorized in two groups: the mechanisms that impose high overhead, but generate an accurate communication pattern, and the mechanisms that are light-weight, but rely on indirect and unreliable information about the communication. We note that are few, or none, solutions that provide an accurate communication pattern with low overhead.

In this master thesis, we presented two new methods to find the communication patterns between threads in shared-memory applications. Our first method is implemented directly into the cache memory subsystem. It makes use of the information about the shared cache lines provided by cache coherence protocols. We developed mechanisms for snoop and directory coherence protocols. Our second method uses the TLB to detect which memory pages each core is accessing. Communication is detected when the same memory page is found in more than one TLB. We developed mechanisms for both software-managed and hardware-managed TLBs, covering most of the current architectures. 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.

In contrast to traditional approaches, our proposals do not require time consuming tasks such as simulation or modifications to the source code of the applications. Our mechanisms presented no time overhead for the cache coherence based mechanisms and a very low time overhead for both TLB types. They rely on hardware features and allow the communication patterns to be discovered dynamically by the operating system during the execution of the application. Furthermore, they are independent from the implementation of the parallel applications. For these reasons, our mechanisms are suitable in real-world scenarios.

We evaluated our proposal using the Simics simulator and applications from the NPB. We were able to identify the communication patterns for all NPB applications. The directory cache coherence based mechanisms are more dependent on the cache size than the snoop ones, since the directory based mechanisms detect the communication using the last private cache level. Nevertheless, they proved to be very effective with a cache size of 256kb, common for current L2 private caches. The software-managed TLB based mechanism presented more accurate results than the hardware-managed one, since the hardware-managed TLB may suffer from bad sampling.

We used the discovered 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 31.9%. 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 cases, 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 communicate more and present heterogeneous communication patterns showed the greatest improvements. Applications that have homogeneous communication patterns did not present improvements, which is the expected result, as there is no difference in the communication among the threads to be exploited.

For the future, we intend to develop a thread migration strategy, so that the mapping can be changed during execution. We also want to improve the approach for the hardware-managed TLB.

REFERENCES

AGARWAL, A. et al. An evaluation of directory schemes for cache coherence. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 15., Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society Press, 1988. p.280–298. (ISCA ’88).

AJIMA, Y.; SUMIMOTO, S.; SHIMIZU, T. Tofu: a 6d mesh/torus interconnect for exascale computers. **Computer**, [S.l.], v.42, n.11, p.36 –40, nov. 2009.

ALVES, M. A. Z.; FREITAS, H. C.; NAVAUX, P. O. A. Investigation of Shared L2 Cache on Many-Core Processors. In: WORKSHOP ON MANY-CORE, Berlin. **Proceedings...** VDE Verlag GMBH, 2009. p.21–30.

AMD. **AMD64 Architecture Programmers Manual Volumes 1, 2 e 3**: application programming, system programming e general-purpose and system instructions. [S.l.]: Advanced Micro Devices, 2007.

AMD. **AMD Opteron 6200 Series Processor**. Available at, http://www.amd.com/us/products/server/processors/6000-series-platform/6200/Pages/6200-series-processors.aspx. 2011.

AWASTHI, M. et al. Handling the problems and opportunities posed by multiple on-chip memory controllers. In: PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 19., New York, NY, USA. **Proceedings...** ACM, 2010. p.319–330. (PACT ’10).

AZIMI, R. et al. Enhancing operating system support for multicore processors by using hardware performance monitoring. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.43, p.56–65, April 2009.

BACH, M. M. et al. Analyzing Parallel Programs with Pin. **Computer**, Los Alamitos, CA, USA, v.43, n.3, p.34–41, 2010.

BARROW-WILLIAMS, N.; FENSCH, C.; MOORE, S. A communication characterisation of Splash-2 and Parsec. In: WORKLOAD CHARACTERIZATION, 2009. IISWC 2009. IEEE INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2009. p.86 –97.

BELLARD, F. QEMU, a fast and portable dynamic translator. In: ATEC ’05: PROCEEDINGS OF THE ANNUAL CONFERENCE ON USENIX ANNUAL TECHNICAL CONFERENCE, Berkeley, CA, USA. **Anais...** USENIX Association, 2005. p.41–41.

BIENIA, C. et al. The PARSEC benchmark suite: characterization and architectural implications. In: PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 17., New York, NY, USA. **Proceedings...** ACM, 2008. p.72–81. (PACT '08).

BIENIA, C.; KUMAR, S.; LI, K. 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. **Anais...** [S.l.: s.n.], 2008. p.47 –56.

BOKHARI, S. On the Mapping Problem. **Computers, IEEE Transactions on**, [S.l.], v.C-30, n.3, p.207 –214, march 1981.

BORKAR, S.; CHIEN, A. A. The future of microprocessors. **Commun. ACM**, New York, NY, USA, v.54, p.67–77, May 2011.

BRAMS. **BRAMS**. Available at, http://www.cptec.inpe.br/brams/. 2009.

BROQUEDIS, F. et al. Structuring the execution of OpenMP applications for multicore architectures. In: PARALLEL DISTRIBUTED PROCESSING (IPDPS), 2010 IEEE INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2010. p.1 –10.

BROQUEDIS, F. et al. hwloc: a generic framework for managing hardware affinities in hpc applications. In: PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP), 2010 18TH EUROMICRO INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2010. p.180 –186.

CHEN, H. et al. MPIPP: an automatic profile-guided parallel process placement toolset for smp clusters and multiclusters. In: SUPERCOMPUTING, 20., New York, NY, USA. **Proceedings...** ACM, 2006. p.353–360. (ICS '06).

CHISHTI, Z.; POWELL, M.; VIJAYKUMAR, T. Optimizing replication, communication, and capacity allocation in CMPs. In: COMPUTER ARCHITECTURE, 2005. ISCA '05. PROCEEDINGS. 32ND INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2005. p.357 – 368.

COTEUS, P. W. et al. Technologies for exascale systems. **IBM Journal of Research and Development**, [S.l.], v.55, n.5, p.14:1 –14:12, sept.-oct. 2011.

CRUZ, E.; ALVES, M.; NAVAUX, P. Process Mapping Based on Memory Access Traces. In: COMPUTING SYSTEMS (WSCAD-SCC), 2010 11TH SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2010. p.72 –79.

CRUZ, E.; DIENER, M.; NAVAUX, P. Using the Translation Lookaside Buffer to Map Threads in Parallel Applications Based on Shared Memory. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS) (APPROVED, AWAITING TO BE PUBLISHED), 26. **Anais...** [S.l.: s.n.], 2012.

CRUZ, E. et al. 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. **Anais...** [S.l.: s.n.], 2011. p.551 –558.

CRUZ, E. et al. Memory-aware Thread and Data Mapping for Hierarchical Multi-core Platforms. **International Journal of Networking and Computing (IJNC)**, [S.l.], v.2, n.1, p.97–116, 2012.

CRUZ, E.; NAVAUX, P. **Técnicas para monitorar acessos à memória para mapeamento estático de processos**. [S.l.]: Instituto de Informática, Universidade Federal do Rio Grande do Sul, 2010. (1357).

DE MICHELI, G.; BENINI, L. **Networks on Chips**: technology and tools. [S.l.]: Morgan Kaufmann, 2006.

DIENER, M. et al. Evaluating Thread Placement Based on Memory Access Patterns for Multi-core Processors. In: IEEE 12TH INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS, 2010., Washington, DC, USA. **Proceedings. . .** IEEE Computer Society, 2010. p.491–496. (HPCC '10).

EGGERS, S. J.; KATZ, R. H. Evaluating the performance of four snooping cache coherency protocols. In: COMPUTER ARCHITECTURE, 16., New York, NY, USA. **Proceedings. . .** ACM, 1989. p.2–15. (ISCA '89).

FREITAS, H. C. et al. Evaluating Network-on-Chip for Homogeneous Embedded Multiprocessors in FPGAs. In: ISCAS: INT. SYMP. ON CIRCUITS AND SYSTEMS. **Proceedings. . .** [S.l.: s.n.], 2007. p.3776–3779.

HAMERLY, G. et al. Simpoint 3.0: faster and more flexible program phase analysis. **Journal of Instruction Level Parallelism**, [S.l.], v.7, n.4, 2005.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture**: a quantitative approach. 4th.ed. USA: Elsevier, 2007.

IBRAHIM, K. Bridging the gap between complex software paradigms and power-efficient parallel architectures. In: GREEN COMPUTING CONFERENCE, 2010 INTERNATIONAL. **Anais. . .** [S.l.: s.n.], 2010. p.417 –424.

INTEL. **IA-32 Intel Architecture Software Developer's Manual Volumes 1, 2 e 3**: basic architecture, instruction set reference e system programming guide. [S.l.]: Intel Corporation, 2004.

INTEL. **Quad-Core Intel Xeon Processor 5400 Series**. [S.l.]: Intel Corporation, 2008.

INTEL. **Intel 64 and IA-32 Architectures Software Developer's Manual Volumes 1, 2 e 3**: basic architecture, instruction set reference e system programming guide. [S.l.]: Intel Corporation, 2009.

INTEL. **Intel Xeon Processor 5600 Series**. [S.l.]: Intel Corporation, 2011.

JIN, H.; FRUMKIN, M.; YAN, J. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. In: TECHNICAL REPORT: NAS-99-011. **Anais. . .** [S.l.: s.n.], 1999.

LIU, Y.; ZHANG, E.; SHEN, X. A cross-input adaptive framework for GPU program optimizations. In: PARALLEL DISTRIBUTED PROCESSING, 2009. IPDPS 2009. IEEE INTERNATIONAL SYMPOSIUM ON. **Anais. . .** [S.l.: s.n.], 2009. p.1 –10.

MA, C. et al. An approach for matching communication patterns in parallel applications. In: PARALLEL DISTRIBUTED PROCESSING, 2009. IPDPS 2009. IEEE INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2009. p.1 –12.

MAGNUSSON, P. S. et al. Simics: a full system simulation platform. **Computer**, Los Alamitos, CA, USA, v.35, p.50–58, 2002.

MARTIN, M. M. K. et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. **SIGARCH Comput. Archit. News**, New York, NY, USA, v.33, p.92–99, November 2005.

MIPS. **MIPS R10000 Microprocessor User's Manual**. [S.l.]: MIPS Technologies, Inc., 1996.

MOGUL, J. C.; BORG, A. The effect of context switches on cache performance. **SIGPLAN Not.**, New York, NY, USA, v.26, p.75–84, April 1991.

MOORE, S.; RALPH, J. User-defined events for hardware performance monitoring. **Procedia Computer Science**, [S.l.], v.4, n.0, p.2096 – 2104, 2011. <ce:title>Proceedings of the International Conference on Computational Science, ICCS 2011</ce:title>.

MPI. **MPI**: a message-passing interface standard. [S.l.]: MPI Forum, 2009.

NETHERCOTE, N.; SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. **SIGPLAN Not.**, New York, NY, USA, v.42, n.6, p.89–100, 2007.

OLUKOTUN, K. et al. The Case for a Single-Chip Multiprocessor. In: INT. SYMP. ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS. **Anais...** IEEE, 1996. p.2–11.

OPENMP. **OpenMP Application Program Interface**. [S.l.]: OpenMP Architecture Review Board, 2008.

OSIAKWAN, C.; AKL, S. 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. **Anais...** [S.l.: s.n.], 1990. p.880 –887.

OTT, M. et al. autopin-Automated Optimization of Thread-to-Core Pinning on Multicore Systems. In: WORKSHOP ON PROGRAMMABILITY ISSUES FOR MULTI-CORE COMPUTERS (MULTIPROG), 1. **Proceedings...** [S.l.: s.n.], 2008.

REINDERS, J. **Intel threading building blocks**. 1.ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

RIBEIRO, C. et al. Memory Affinity for Hierarchical Shared Memory Multiprocessors. In: COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 2009. SBAC-PAD '09. 21ST INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2009. p.59 –66.

RIBEIRO, C. P. et al. Improving Memory Affinity of Geophysics Applications on NUMA platforms Using Minas. In: INTERNATIONAL MEETING HIGH PERFORMANCE COMPUTING FOR COMPUTATIONAL SCIENCE, VECPAR, 9., US. **Anais...** LNCS, 2010.

RODRIGUES, E. et al. Multi-core aware process mapping and its impact on communication overhead of parallel applications. In: COMPUTERS AND COMMUNICATIONS, 2009. ISCC 2009. IEEE SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2009. p.811 –817.

SAITO, H. et al. Large System Performance of SPEC OMP2001 Benchmarks. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTING, 4., London, UK, UK. **Proceedings...** Springer-Verlag, 2002. p.370–379. (ISHPC '02).

SCOTCH. **Scotch**. Available at, http://www.labri.fr/perso/pelegrin/scotch/. 2010.

SEBESTA, R. W. **Concepts of Programming Languages**. 9th.ed. USA: Addison-Wesley Publishing Company, 2009.

SHALF, J.; DOSANJH, S.; MORRISON, J. Exascale computing technology challenges. In: HIGH PERFORMANCE COMPUTING FOR COMPUTATIONAL SCIENCE, 9., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2011. p.1–25. (VECPAR'10).

SHWARTSMAN, S.; MIHOCKA, D. Virtualization Without Direct Execution or Jitting: designing a portable virtual machine infrastructure. In: THE 35TH INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE (ISCA), Beijing, China. **Anais...** [S.l.: s.n.], 2008.

SONNEK, J. et al. Starling: minimizing communication overhead in virtualized computing platforms using decentralized affinity-aware migration. In: INT. CONF. ON PARALLEL PROCESSING (ICPP). **Anais...** [S.l.: s.n.], 2010. p.228 –237.

SPARC. **The SPARC Architecture Manual Version 9**. [S.l.]: SPARC International, Inc., 2000.

STALLINGS, W. **Computer Organization and Architecture**: designing for performance. [S.l.]: Prentice Hall, 2006.

STEVENSON, D.; CONN, R. Bridging the Interconnection Density Gap for Exascale Computation. **Computer**, [S.l.], v.44, n.1, p.49 –57, jan. 2011.

STREAM. **Stream**. Available at, http://www.cs.virginia.edu/stream/. 2011.

TAM, D.; AZIMI, R.; STUMM, M. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.41, p.47–58, March 2007.

TAM, D.; AZIMI, R.; STUMM, M. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In: ACM SIGOPS/EUROSYS EUROPEAN CONFERENCE ON COMPUTER SYSTEMS 2007, 2., New York, NY, USA. **Proceedings...** ACM, 2007. p.47–58. (EuroSys '07).

TANENBAUM, A. S. **Modern Operating Systems**. 3rd.ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

TERBOVEN, C. et al. Data and thread affinity in openmp programs. In: MEMORY ACCESS ON FUTURE PROCESSORS: A SOLVED PROBLEM?, 2008., New York, NY, USA. **Proceedings...** ACM, 2008. p.377–384. (MAW '08).

TORRELLAS, J. Architectures for Extreme-Scale Computing. **Computer**, [S.l.], v.42, n.11, p.28 –35, nov. 2009.

TULLSEN, D. M.; EGGERS, S. J.; LEVY, H. M. Simultaneous multithreading: maximizing on-chip parallelism. In: COMPUTER ARCHITECTURE, 22., New York, NY, USA. **Proceedings...** ACM, 1995. p.392–403. (ISCA ’95).

WANG, Z.; O’BOYLE, M. F. Mapping parallelism to multi-cores: a machine learning based approach. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 14., New York, NY, USA. **Proceedings...** ACM, 2009. p.75–84. (PPoPP ’09).

WOO, S. C. et al. The SPLASH-2 programs: characterization and methodological considerations. In: COMPUTER ARCHITECTURE, 22., New York, NY, USA. **Proceedings...** ACM, 1995. p.24–36. (ISCA ’95).

ZHAI, J. et al. Efficiently Acquiring Communication Traces for Large-Scale Parallel Applications. **Parallel and Distributed Systems, IEEE Transactions on**, [S.l.], v.22, n.11, p.1862 –1870, nov. 2011.

ZHAO, H.; SHRIRAMAN, A.; DWARKADAS, S. SPACE: sharing pattern-based directory coherence for multicore scalability. In: PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 19., New York, NY, USA. **Proceedings...** ACM, 2010. p.135–146. (PACT ’10).

ZHOU, X.; CHEN, W.; ZHENG, W. Cache Sharing Management for Performance Fairness in Chip Multiprocessors. In: PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 2009. PACT ’09. 18TH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2009. p.384 –393.

7 APPENDIX - SUMMARY IN PORTUGUESE

In this chapter, we present a summary of this master thesis in the portuguese language, as required by the PPGC Graduate Program in Computing.

Neste capítulo, é apresentado um resumo desta dissertação de mestrado na língua portuguesa, como requerido pelo Programa de Pós-Graduação em Computação.

7.1 Introdução

A melhoria da capacidade de processamento ocorria tradicionalmente através do aumento da frequência de processadores. Entretanto, o consumo de potência e problemas de atraso do fio, bem como a dificuldade no aumento de estágios do *pipeline*, têm guiado as indústrias de microprocessadores de propósito geral a apostarem na integração de múltiplos núcleos de processamento dentro do chip (*multi-core*), sustentando assim a lei de Moore (OLUKOTUN et al., 1996). Esses processadores focam na extração de paralelismo no nível de fluxo de execução de múltiplas tarefas, gerando assim um ambiente propício para o aumento de desempenho de aplicações paralelas.

Dentro da arquitetura desses processadores *multi-core*, o subsistema de memória se apresenta como provedor de dados para os diversos núcleos de processamento. Dessa maneira, a hierarquia de memória *cache* desses processadores *multi-core* tem sofrido constantes mudanças para melhor se adaptar às necessidades computacionais, escondendo assim a lacuna de desempenho entre memória e processador (HENNESSY; PATTERSON, 2007). Tal lacuna de desempenho é conhecida por *memory wall*. Nas arquiteturas de larga escala, o *memory wall* é agravado pelo alto número de núcleos de processamento, fazendo-se necessário soluções inovadoras para que o sistema escale e apresente um custo energético viável (COTEUS et al., 2011; TORRELLAS, 2009).

Um dos principais problemas encontrados nas arquiteturas de larga escala é a comunicação entre as tarefas (ZHAI et al., 2011). A comunicação implica na movimentação de dados entre os núcleos de processamento, acarretando perda de desempenho e gasto e energia (BORKAR; CHIEN, 2011). Nessas arquiteturas de larga escala, há diversas camadas de memória *cache*. A troca de informações entre diferentes *threads* de um mesmo programa paralelo pode variar dependendo do núcleo de processamento que determinada *thread* está executando. Esta diferença introduzida pela memória *cache* varia quando grupo de núcleos de processamento compartilham uma mesma memória *cache*. Se forem consideradas ainda as futuras topologias introduzidas pelas NoC (*Network-on-Chip*)(DE MICHELI; BENINI, 2006), os custos de troca de informação entre *threads* poderá sofrer maiores variações (FREITAS et al., 2007). Nesse dado cenário, o mapeamento de *threads* possibilita um aumento de desempenho ao se mapear as *threads* levando em consideração a memória compartilhada entres as *threads*.

O objetivo deste trabalho é propor técnicas dinâmicas de mapeamento de *threads* a fim de melhorar o desempenho de aplicações paralelas. O mapeamento deverá levar em consideração a topologia da memória *cache* e os custos de troca de dados entre processadores e núcleos de processamento. As propostas foram implementadas no simulador Simics (MAGNUSSON et al., 2002) e, para avaliação de desempenho, foi utilizado um ambiente real multiprocessado. Como carga de trabalho, o conjunto de aplicações cientificas paralelas NAS-NPB (JIN; FRUMKIN; YAN, 1999) foi utilizado para comparar o desempenho do mapeamento automático do sistema operacional com o mapeamento otimizado fornecido por nossas propostas.

O restante deste capítulo está organizado da seguinte forma. A Seção 7.2 fala sobre o mapeamento de *threads*. A Seção 7.3 apresenta as propostas de mecanismos de mapeamento dinâmico de *threads*. A Seção 7.4 mostra a metodologia empregada. A Seção 7.5 apresenta os resultados. Por fim, a Seção 7.6 contém a conclusão e trabalhos futuros.

7.2 Mapeamento de Threads

O mapeamento de *threads* é uma técnica que possibilita o aumento de desempenho em aplicações paralelas através de uma alocação mais eficiente dos recursos, neste caso, a hierarquia de memória. É necessário coletar informações sobre o padrão de comunicação das aplicações, que tem uma dificuldade variável dependendo do paradigma de programação paralela adotado. Quando utilizado paradigmas de programação orientados a passagem de mensagens (RODRIGUES et al., 2009), a descoberta do padrão de comunicação é trivial, já que a comunicação é explícita. Entretanto, com o uso de programação paralela para memória compartilhada em ambientes multiprocessados, a tarefa de descoberta do padrão de comunicação se torna mais difícil, pois a comunicação é implícita e ocorre através do acesso a regiões de memória compartilhadas por diferentes *threads*.

Escalonando as *threads* que compartilham memória em núcleos que compartilham uma mesma memória *cache* propicia um melhor desempenho de que quando nenhuma memória *cache* é compartilhada (ALVES; FREITAS; NAVAUX, 2009). Além disso, o tempo para dois núcleos dentro de um mesmo *chip* se comunicarem é menor do que quando as *threads* encontram-se em processadores separados. Essas diferença no desempenho se devem ao fato de que além de um melhor aproveitamento de espaço nas memórias *cache*, um menor número de invalidações deverá ocorrer a cada modificação dos dados. Dessa forma, apenas os níveis superiores, mais próximos do processador, devem receber os valores atualizados, reduzindo assim a sobrecarga imposta por protocolos de coerência. Dessa forma, as *threads* que mais compartilham memória devem ser escalonadas em núcleos mais próximos em relação à hierarquia de memória adotada.

Em relação à arquiteturas com características de acesso não-uniforme à memória (NUMA), além de mapeamento de *thread*, também é importante mapear os dados (RIBEIRO et al., 2009). O mapeamento de dados é necessário em máquinas NUMA porque a latência de acesso aos bancos de memória são diferentes entre os núcleos. Os núcleos são divididos em grupos, em que cada grupo é um nó NUMA. Cada nó NUMA tem seus bancos de memória próprios. Quando um acesso é realizado a um banco de memória localizado no mesmo nó NUMA, este tipo de acesso é denominado acesso local. Quando um acesso é realizado a um banco de memória de outro nó NUMA, o acesso é denominado remoto.

Um ponto importante a ser levado em consideração é a usabilidade dos modelos propostos. Arquiteturas como as de processamento gráfico apresentam alto desempenho, porém sua programação é complexa (LIU; ZHANG; SHEN, 2009), desencorajando seu uso por parte de muitos desenvolvedores de *software*. Algumas técnicas exigem etapas custosas de *profiling*, desencorajando o uso das mesmas (WANG; O'BOYLE, 2009). Outras necessitam que os programadores das aplicações insiram anotações no código fonte ou , aumentando a complexidade da programação (IBRAHIM, 2010). Tais técnicas baseadas em anotações no código-fonte podem diminuir sua portabilidade, já que frequentemente as anotações são dependentes da arquitetura alvo. Além disso, depender de anotações no código-fonte diminui a confiabilidade do sistema, pois programadores inexperientes podem inserir anotações equivocadas.

O mapeamento de *threads* pode ser estático ou dinâmico. No mapeamento estático, análises prévias podem ser feitas com a aplicação. Etapas custosas, como simulação, são necessárias para monitorar os acessos à memória e descobrir o padrão de compartilhamento (CRUZ; ALVES; NAVAUX, 2010). No mapeamento dinâmico, o padrão de compartilhamento deve ser descoberto durante a execução da aplicação. Ele deve apresentar baixa sobrecarga e não pode interferir no comportamento da aplicação. Técnicas atuais de mapeamento dinâmico de *threads* (TAM; AZIMI; STUMM, 2007a,b; AZIMI et al., 2009; BROQUEDIS et al., 2010) não proveem informações precisas sobre o padrão de comunicação.

7.3 Propostas de mecanismos para mapeamento dinâmico de threads

Nesta seção, são explicados os mecanismos propostos para o mapeamento dinâmico de *threads*. As propostas podem classificados em duas categorias principais: mecanismos baseados em coerência de cache e mecanismos baseados em TLB.

7.3.1 Mecanismos baseados em coerência de cache

Protocolos de coerência de cache são responsáveis por manter a integridade dos dados em arquiteturas onde há mais de uma memória cache presente, como é usual em ambientes multicore ou multiprocessados. Estes protocolos mantêm informações sobre se uma linha é privada ou compartilhada entre duas ou mais caches. Isto pode ser explorado a fim de estimar a quantidade de comunicação entre as *threads*, uma vez que o acesso a uma linha compartilhada por duas ou mais caches representa uma comunicação. Pequenas modificações nos protocolos são necessárias para identificar os padrões de compartilhamento de dados. Neste trabalho, são propostas duas modificações, uma para protocolos baseados em espionagem e uma para protocolos baseados em diretório.

Ambos os mecanismos baseados em espionagem e diretório partem de um mesmo princípio, que consiste de que, em protocolos MESI e seus derivados, uma mensagem de invalidação é transmitida quando uma escrita é feita em uma linha de cache compartilhada ou inválida. No entanto, nenhuma mensagem é transmitida quando uma leitura é realizada em uma linha de cache compartilhada. Por isso, apenas as mensagens de invalidação são monitoradas. A comunicação é detectada quando as linhas de cache que tem seu estado como compartilhada são invalidadas. Foram desenvolvidos 2 mecanismos baseados em espionagem e 2 mecanismos baseados em diretório.

7.3.1.1 Mecanismos baseados em espionagem

O primeiro mecanismo baseado em espionagem requer memórias cache privadas. É adicionado uma matriz para cada cache para armazenar a comunicação em relação ao núcleo que está ligado à cache. O número de linhas dessa matriz é igual ao número total de núcleos, e o número de colunas é um. A Figura 3.1 contém um esquemático do funcionamento, e a Tabela 3.1 contém alguns exemplos. Quando uma escrita é feita em uma linha de cache compartilhada, como no Exemplo 3, é enviada uma mensagem de invalidação junto com o ID do núcleo que enviou a mensagem no barramento de espionagem. As memória caches que invalidam uma linha em resposta a esta mensagem incrementam sua matriz de comunicação na célula cuja linha é o ID do núcleo que iniciou a transação, contido no barramento. O mesmo acontece quando uma escrita é feita em uma linha de cache inválida, como no Exemplo 2.

O protocolo explicado no parágrafo anterior se limita a caches privadas. O segundo protocolo baseado em espionagem supera este problema, mas requer mais modificações no hardware. A fim de determinar qual o núcleo acessou uma linha de cache quando houver mais de um núcleo compartilhando uma mesma cache, é adicionado um bit de acesso (A-Bit) por núcleo que compartilha a cache em cada linha de cache. Estes bits de acesso mostram quais dos núcleos acessaram cada linha de cache. É adicionado uma matriz para cada cache, com número de linhas da matriz igual ao número total de núcleos, e número de colunas igual ao número de núcleos compartilhando a cache. A Figura 3.2 mostra um esquemático do segundo mecanismo baseado em espionagem, e a Tabela 3.2 contém alguns exemplos.

Quando qualquer solicitação chega na cache, a cache seta o bit A-Bit referente ao núcleo que iniciou a operação na linha de cache solicitada, como no Exemplo 1. Se um acerto de escrita ocorre, a matriz de comunicação da cache é atualizada nas células cuja linha é o ID do núcleo que iniciou a transação, e colunas onde o A-Bits são definidos (Exemplo 2). Isto é feito para detectar a comunicação entre os núcleos que compartilham a cache. Além disso, se um acerto de escrita acontece em uma linha de cache compartilhada, uma mensagem de invalidação é transmitida junto com o ID do núcleo que iniciou a transação. As caches que invalidam uma linha em resposta a esta mensagem incrementam suas matrizes de comunicação nas células cuja linha é o ID do núcleo contido no barramento, e colunas cujos A-Bits estão setados, que é detalhado no Exemplo 3. É importante notar que, quando mais de 1 A-Bit está setado, mais de uma célula é atualizada, como no Exemplo 4. A invalidação também é transmitido junto com o ID do núcleo quando uma escrita é feita em uma linha de cache inválida.

7.3.1.2 Protocolos baseados em diretório

Em protocolos de coerência de cache baseados em diretório, o diretório já mantém informações sobre quais caches estão compartilhando cada linha de cache, que pode ser visto no Exemplo 1 da Tabela 3.3. A única modificação necessária para o hardware é a adição de uma matriz para cada diretório. O número de linhas e colunas da matriz é igual ao número total de núcleos.

É importante mencionar que o diretório mantém informações de quais caches estão compartilhando cada linha, não quais núcleos estão compartilhando cada linha. Portanto, tem-se que usar as informações sobre o último nível privado de cache para detectar se um núcleo está acessando determinada linha de cache, uma vez que os níveis compartilhados de cache podem ser acessado por mais de um núcleo. Por exemplo, considerando uma arquitetura com 2 níveis de cache, e apenas acache L1 é privada, não é possível inferir qual núcleo acessou cada linha da cache L2. No entanto, as linhas presentes no cache L1 foram certamente acessada pelo núcleo correspondente.

Em alguns protocolos baseados em diretório, o diretório mantém informações de quais processadores estão compartilhando cada linha (ZHAO; SHRIRAMAN; DWARKADAS, 2010). O diretório apenas encaminha os pedidos para os processadores que tem a linha de cache correspondente. É responsabilidade de cada processador manter informações sobre quais caches internas ao mesmo compartilham cada linha de cache. A única modificação necessária para o hardware é a adição uma matriz para cada cache. O número de linhas da matriz é igual ao número total de núcleos, e o número de colunas é igual ao número de núcleos compartilhando a cache.

Em geral, os mecanismos baseados em diretório funcionam da mesma maneira que os de espionagem. A diferença mais relevante é que os baseados em espionagem detectam a comunicação nos *broadcasts* de mensagens de invalidação, já os baseados em diretório detectam a comunicação quando o diretório é acessado para se descobrir quais caches partilham a linha correspondente.

7.3.2 Mecanismos baseados em TLB

A memória virtual exige a tradução de endereços virtuais para endereços físicos em cada acesso à memória. Para isso, o sistema operacional mantém tabelas na memória principal que fazem essa tradução possível. Estas tabelas de tradução contêm o endereço físico para cada página virtual, e são indexados pelos bits superiores do endereço virtual para tradução rápida. No entanto, os acessos à memória realizados na tabela de tradução impõe uma sobrecarga alta, e algumas arquiteturas ainda exigem vários acessos, no caso da tabela de tradução ser composta por mais de um nível. Para superar esses problemas, uma memória cache especial, chamada *Transation Lookaside Buffer (TLB)*, é responsável por armazenar as entradas da tabela de tradução mais acessadas recentemente.

Em arquiteturas multicore, cada núcleo tem sua própria TLB, que armazena as entradas da tabela mais recentemente acessadas pelo núcleo. Se uma determinada entrada da tabela está presente em mais de uma TLB, significa que os núcleos correspondentes acessaram uma região de memória compartilhada, pelo menos na granularidade de nível de página. Se todas as TLBs forem varridas e registradas cada vez que uma entrada é partilhada por mais de uma TLB, temos a quantidade de páginas compartilhadas pelos núcleos como resultado. Sistematicamente fazendo este procedimento, obtemos uma representação do padrão de comunicação na granularidade de nível de página. Esse padrão pode ser usado para mapear as *threads* das aplicações em arquiteturas multicore.

Arquiteturas atuais de processadores gerenciam a TLB de diferentes maneiras. Os dois tipos mais importantes de gerência requerem métodos ligeiramente diferentes para descobrir o padrão de comunicação.

7.3.2.1 TLBs gerenciadas via software

Em algumas arquiteturas RISC, como na SPARC (SPARC, 2000), o processador desvia para o sistema operacional quando uma falta ocorre na TLB. O sistema operacional acessa a tabela da página na memória principal e carrega a entrada correspondente na TLB. Este tipo de TLB é chamado de TLB gerenciada via *software*. As principais vantagens deste tipo de gestão é que ele simplifica o *hardware* e é muito flexível, já que o sistema operacional pode escolher a forma de implementar a memória virtual.

Para implementar um mecanismo para detectar o padrão de comunicação usando a TLB gerenciada via software, nenhuma modificação de *hardware* é necessária. Quando uma falta na TLB gera um desvio para o sistema operacional, o *kernel* pode também verificar todas as TLBs para procurar por entradas iguais, além de carregar a entrada da memória principal. Acessar a TLB de outros núcleos pode representar um gargalo. Para superar este problema, o conteúdo de todas as TLBs pode ser espelhado na memória principal. Isso não exigiria muito espaço de armazenamento, pois o tamanho da TLB é geralmente pequeno para manter a latência de acesso de baixo. Para reduzir ainda mais o impacto da iteração sobre as TLBs, o sistema operacional poderia tratar a TLB como uma cache associativa por conjuntos, de modo que apenas algumas entradas de cada TLB têm de ser comparados. Além disso, em vez de correr a busca em todas as faltas na TLB, a busca pode ser executado em apenas uma fração deles. Isso diminui a precisão, mas reduz a sobrecarga pela mesma fração utilizada para a amostragem.

7.3.2.2 TLBs gerenciadas via hardware

Arquiteturas como a x86 e x86-64 (INTEL, 2009) mantêm a TLB como um cache para as entradas de tabela de página armazenados na memória principal. Para cada acesso à memória, a TLB é pesquisada. Se a entrada correspondente está na TLB, o endereço é traduzido e enviado para a hierarquia de memória. Se a entrada não está presente na TLB, o *hardware* acessa a memória principal e carrega a entrada correspondente na TLB. Este mecanismo é chamado TLB gerenciada via *hardware*. O sistema operacional só mantém o conteúdo da tabela de página na memória principal. A única operação que é realizada pelo sistema operacional neste tipo de TLB é invalidar as entradas quando a tabela da página é modificada. A gerência via *hardware* tem um baixo impacto no desempenho, pois não requer desvios e trocas de contexto em cada falta na TLB.

Para permitir encontrar o padrão de comunicação, arquiteturas com TLB gerenciada via *hardware* requerem uma pequena alteração no hardware, já que o sistema operacional não tem acesso ao conteúdo da TLB. A modificação consiste em adicionar uma instrução que permite ao sistema operacional acessar o conteúdo da TLB. Desta forma, o *kernel* pode varrer as TLBs para encontrar entradas correspondentes periodicamente. A precisão e sobrecarga desse mecanismo depende do tempo entre as pesquisas.

7.4 Metodologia

As aplicações são executadas dentro do simulador para detectar os padrões de comunicação. Foi utilizado o simulador Simics (MAGNUSSON et al., 2002) e o modelo de memória GEMS/Ruby (MARTIN et al., 2005). Para avaliar o desempenho, usamos o padrão de comunicação obtidos no simulador para mapear as *threads* em uma máquina real. É importante notar que as *threads* ainda não são migradas durante a execução das aplicações. A migração dinâmica requer um algoritmo para detectar quando o padrão muda de comunicação (MA et al., 2009), bem como modificações no escalonador do sistema operacional. Estes pontos estão fora do escopo deste trabalho, que é apresentar mecanismos que dinamicamente detectar os padrões de comunicação. As máquina utilizadas nos experimentos contém 2 processadores, cada um com 4 núcleos, totalizando 8 núcleos de execução, sendo que a memória *cache* L2 é compartilhada por cada par de núcleos.

O algoritmo utilizado para mapear as *threads* nos núcleos da máquina real se baseia na teoria dos grafos. O mapeamento é modelado como um problema de emparelhamento máximo de custo mínimo em grafos. Tal problema consiste de: dado um grafo $G = (V, E)$, deseja-se achar um subconjunto M de E no qual todos os vértices $v \in V$ incidem em no máximo um elemento de M e que as somas dos pesos das arestas seja mínimo. O grafo pode ser montado diretamente a partir da matriz de compartilhamento, basta considerar cada *thread* um vértice e a quantidade de memória compartilhada o peso da aresta, gerando um grafo completo. Em (OSIAKWAN; AKL, 1990), é descrito um algoritmo paralelo para se encontrar emparelhamentos perfeitos de custo máximo em grafos valorados completos que possui uma complexidade temporal de $O(\frac{N^3}{P} + N^2 \lg N)$, onde N é o número de vértices (*threads*) e P é o número de processadores.

Para um melhor entendimento do desempenho, além do tempo de execução, foram medido alguns eventos na máquina real através de contadores de *hardware* (INTEL, 2009). Foram contados o número de mensagens de invalidação, faltas na cache L2 e o número de transações de espionagem. Para acessar os contadores de *hardware*, a biblioteca PAPI (MOORE; RALPH, 2011) foi empregada.

As aplicações utilizadas nos experimentos fazem parte da carga de trabalho *Numerical Aerodynamic Simulation Parallel Benchmark* (NAS-NPB) versão 3.3.1, paralelizada com OpenMP. Essa carga de trabalho é formada por diversas aplicações relacionadas a métodos numéricos de simulações aerodinâmicas para computação científica. Esses aplicativos foram projetados para comparar o desempenho de computadores paralelos, sendo formados por *kernels* e problemas de simulação de dinâmica de fluídos computacionais (JIN; FRUMKIN; YAN, 1999). O NAS-NPB conta com diversos tamanhos de entrada para os problemas, sendo que para este trabalho foi utilizado o tamanho W, que é o mais indicado para simulações.

7.5 Resultados

Nesta seção, apresentamos os resultados obtidos pelos mecanismos propostos sobre utilizando *benchmarks*. Primeiro, são mostrados os padrões de comunicação descobertos, que são comparados à linha de base, descrita na Seção 4.5.2. Depois, são mostrados testes de desempenho mapeando as *threads* utilizando os padrões de comunicação. A partir de agora, a abordagem baseada em espionagem será denominada por *SNOOP*, a baseada em diretório *DIRECTORY*, a baseada em TLB gerenciada via *software SM*, e a abordagem baseada em TLB gerenciada via *hardware HM*.

7.5.1 Padrões de comunicação

As Figuras 5.1 e 5.2 mostram os padrões de comunicação das aplicações do NPB para o SM e HM, respectivamente. Para o SM, em apenas 1% das faltas na TLB foi realizado o procedimento de detectar a comunicação de forma a se minimizar a sobrecarga. O HM foi avaliado utilizando-se um tempo de 10 milhões de ciclos entre cada chamada ao procedimento de detectar a comunicação. A Figura 5.3 mostra o padrão de comunicação da SNOOP. O DIRECTORY foi avaliado com dois tamanhos de cache diferentes para investigar a hipótese de que o mesmo é muito sensível ao tamanho da cache. As Figuras 5.4 e 5.5 mostram o padrão de comunicação do DIRECTORY com caches de 32kb e 256kb, respectivamente.

As aplicações BT, IS, LU, MG, SP e UA apresentaram, na maioria dos mecanismos, um padrão heterogêneo de comunicação. Heterogêneo significa que há diferenças entre a quantidade de comunicação entre as *threads*. O padrão predominante detectado foi o de decomposição de domínio, muito comum em aplicações paralelas, no qual as *threads* dividem os dados de entrada entre si, se comunicando nas bordas de cada subdomínio. Isto fica evidente porque a maior parte das comunicações são feitas entre as *threads* vizinhas. A aplicação LU se diferencia por também apresentar comunicação entre as *threads* mais distantes, apesar de nem todos os mecanismos terem detectado isto. Nas aplicações CG, EP e FT, a comunicação é mais homogênea, isto é, as *threads* apresentam aproximadamente a mesma quantidade de comunicação entre si. Aplicações com padrão heterogêneo de comunicação possuem um maior potencial de ganho de desempenho através do mapeamento que as aplicações homogêneas.

Sobre os mecanismos baseado em TLB, em geral, os padrões de comunicação detectados pelo SM é mais preciso. A razão é que o mecanismo de descoberta de padrões é capaz de acessar mais amostras do que o HM, já que todas as faltas TLB são tratadas pelo sistema operacional. O HM sofre com o comportamento imprevisível de tempo de execução e amostras ruim. Por exemplo, considerando o caso de que a amostragem é feita quando as *threads* 0 e 1 estão acessando seus dados compartilhados, mas ao mesmo tempo, as outras *threads* estão trabalhando em seus dados privados. Esta situação descreve um comportamento temporário, mas não pode caracterizar o comportamento global da aplicação. Se isso acontecer várias vezes, o HM poderia detectar muita comunicação entre as *threads* 0 e 1, mas nenhuma comunicação entre as outras *threads*. Isto não implica que as outras *threads* não se comunicam entre si, isso significa que a amostragem foi realizada em uma hora inconveniente.

Sobre os mecanismos de coerência cache, pode-se notar que o SNOOP detectou padrões de decomposição de domínio mais expressivos que o DIRECTORY. Esta diferença é causada porque os Bits de acesso são manipulados de forma diferente entre o SNOOP e DIRECTORY. Além disso, como esperado, o diretório é mais preciso com tamanhos maiores cache.

7.5.2 Testes de desempenho

Os resultados apresentados nesta seção são valores médios obtidos por 100 execuções. Para o mecanismo de DIRECTORY, foram utilizados os padrões de comunicação obtidos considerando um tamanho de cache de 256kb. As Figuras 5.6, 5.7, 5.9 e 5.8 apresentam o tempo de execução, número de invalidações, transações de espionagem e faltas na cache L2, respectivamente. Os resultados destes gráficos são normalizados para o tempo do escalonador original do sistema operacional, denotado por OS. A Tabela 5.2 contém os valores absolutos para o número de invalidações, transações de espionagem e faltas na cache, todos divididos pelo tempo de execução. A Tabela 5.3 mostra os desvios padrões.

Em todos os casos, o desempenho foi melhor em comparação com o programador do sistema operacional. O tempo de execução foi melhorado em até 15,3%, e o número de faltas na cache cache foi reduzido em até 31,9%. As invalidações em linhas de cache e transações de espionagem foram reduzidas em até 41% e 65,4%, respectivamente. Isso mostra que o mapeamento de *threads* permite um uso muito melhor das interconexões de arquiteturas atuais. Além disso, a variabilidade dos resultados foi reduzida na maioria dos experimentos, tornando o desempenho dos aplicativo mais previsível.

As melhorias foram dependentes da forma como as aplicações utilizam a memória compartilhada. Como esperado, os aplicativos que mais se comunicam e apresentam padrões de comunicação heterogênea mostraram as maiores melhorias, sendo elas a BT, LU, MG, SP e UA. Aplicações que têm padrões de comunicação homogêneos quase não apresentaram melhorias, que é o resultado esperado, já que não há diferença na comunicação entre as *threads* a serem exploradas, como no CG, EP e FT. Em relação ao IS, a matriz de comunicação não é homogênea. No entanto, o tempo de execução da IS é muito baixo e, apresentando desvios padrões elevados.

Os resultados também mostram que o número de invalidações e transações de espionagem é muito mais sensível ao mapeamento de *threads* que o número de faltas na cache, já que este último apresentou desvios padrões bem mais acentuados. A razão é que um melhor mapeamento influencia diretamente o número de invalidações e transações de espionagem, enquanto que o número de faltas na cache são influenciados por outros fatores, tais como pré-buscas de linhas de cache, a concorrência pelas linhas de cache pelos núcleos que compartilham a cache, entre outros. Da mesma forma, o número de invalidações e transações de espionagem também é mais sensível para o mapeamento de *threads* o tempo de execução.

7.6 Conclusão

Nesta dissertação de mestrado, apresentamos dois novos métodos para encontrar os padrões de comunicação entre as *threads* de aplicações paralelas baseadas em memória compartilhada. O primeiro método é implementado diretamente no subsistema de memória cache. Ele faz uso de informações de compartilhamento das linhas de cache fornecidas pelo sistema de coerência de cache. Foram desenvolvidos mecanismos para protocolos de coerência baseados em espionagem e diretório. O segundo método usa a TLB para detectar quais páginas de memória cada núcleo está acessando. A comunicação é detectada quando uma mesma página de memória é encontrada em mais de uma TLB. Foram desenvolvidos mecanismos para TLBs gerenciadas tanto via *software* quanto *hardware*. Dessa forma, foram cobertas a maior parte das arquiteturas atuais.

Em contraste com as abordagens tradicionais, as propostas realizadas neste trabalho não exigem tarefas demoradas, como a simulação ou modificações ao código fonte das aplicações. Os mecanismos baseados em coerência de cache apresentaram nenhuma sobrecarga de tempo. Os mecanismos baseados em TLB apresentaram uma sobrecarga de tempo muito baixa para ambos os tipos TLB. Eles dependem recursos de hardware e permitem que os padrões de comunicação a serem descobertos dinamicamente pelo sistema operacional durante a execução da aplicação. Além disso, eles são independentes da implementação das aplicações paralelas. Por estas razões, os mecanismos propostos são adequados em cenários do mundo real.

As propostas foram utilizando o simulador Simics e aplicações do NPB, e foram capazes de identificar os padrões de comunicação para todas as aplicações NPB. Os mecanismos baseados em coerência de cache de diretório são mais dependentes do tamanho do cache do que os baseados em espionagem, uma vez que os mecanismos de diretório detectam a comunicação usando o último nível privado de cache. No entanto, eles provaram ser muito eficazes com um tamanho de cache de 256kb, comum para caches L2 atuais privadas. O mecanismo baseado em TLB gerenciada via *software* apresentou resultados mais precisos do que o baseado em TLB gerenciada via *hardware*, pois o segundo pode sofrer de amostragens ruins.

Os padrões de comunicação descobertos foram usados para mapear as *threads* e realizar experimentos de desempenho. Foram medidos o tempo de execução, número de invalidações das linhas de cache, transações de espionagem e faltas na cache, sendo que os resultados foram comparados ao escalonador original do sistema operacional. O tempo de execução foi melhorado em até 15,3%, e o número de faltas na cache foi reduzido em até 31,9%. As invalidações em linhas de cache e transações de espionagem foram reduzidas em até 41% e 65,4%, respectivamente. Isso mostra que o mapeamento *threads* permite um uso muito melhor das interconexões de arquiteturas atuais.

Em todos os casos, o desempenho foi melhorado em comparação com o programador do sistema operacional. Além disso, a variabilidade dos resultados foi reduzida na maioria dos experimentos, tornando o desempenho do aplicativo mais previsível. Melhorias foram dependentes da forma como as aplicações utilizam a memória compartilhada. Como esperado, os aplicativos que mais se comunicam e apresentam padrões de comunicação heterogênea mostraram as maiores melhorias. Aplicações que têm padrões de comunicação homogêneos quase não apresentaram melhorias, que é o resultado esperado, já que não há diferença na comunicação entre as *threads* a serem exploradas.

Para o futuro, pretende-se desenvolver uma estratégia de migração de *threads*, para que o mapeamento possa ser alterado em diferentes fases da execução. Objetiva-se também melhorar a precisão do mecanismo baseado em TLB de *hardware*.