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

EDUARDO HENRIQUE MOLINA DA CRUZ

Online Thread and Data Mapping Using the Memory Management Unit

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

Advisor: Prof. Dr. Philippe O. A. Navaux

Porto Alegre
March 2016

CIP – CATALOGING-IN-PUBLICATION

Cruz, Eduardo Henrique Molina da

Online Thread and Data Mapping Using the Memory Management Unit / Eduardo Henrique Molina da Cruz. – Porto Alegre: PPGC da UFRGS, 2016.

154 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2016. Advisor: Philippe O. A. Navaux.

1. Data movement. 2. Thread and data mapping. 3. Cache memory. 4. NUMA. I. Navaux, Philippe O. A.. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretor do Instituto de Informática: Prof. Luis da Cunha Lamb
Coordenador do PPGC: Prof. Luigi Carro
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"Logic will get you from A to B.
Imagination will take you everywhere."

— ALBERT EINSTEIN

AGRADECIMENTOS

Primeiramente, agradeço a minha mãe e meu pai por todo suporte. Agradeço também a meu primo Gustavo e minha tia Lourdes por todas as visitas e momentos descontraídos. Ao meu tio Luiz, obrigado por toda ajuda disponibilizada. Também não posso esquecer dos amigos Guilherme e Ramon, que também considero como da minha família.

Em relação aos amigos aqui do GPPD, um obrigado inicial ao professor Navaux, pela orientação e compreensão. Em segundo lugar, nada mais nada menos que o grande Matthias, grande parceiro de pesquisa. Não posso esquecer do Laércio, que também muito contribuiu nesta pesquisa. Um abraço também para os outros membros do GPPD, em especial Marco, Francis, Emmanuell e Roloff.

Aos familiares e amigos aqui não mencionados, obrigado por tudo.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS 11

LIST OF FIGURES 13

LIST OF TABLES 17

ABSTRACT 19

RESUMO 21

1 INTRODUCTION 23
1.1 Improving Memory Locality With Sharing-Aware Mapping 24
1.2 Monitoring Memory Accesses for Sharing-Aware Mapping 27
1.3 Using the Memory Management Unit to Gather Information on Memory Accesses 29
1.4 Contributions of this Thesis 31
1.5 Organization of the Text 32

2 SHARING-AWARE MAPPING: OVERVIEW AND RELATED WORK 35
2.1 Improving Memory Locality With Sharing-Aware Mapping 35
2.1.1 Memory Locality in Different Architectures 35
2.1.2 Parallel Applications and Sharing-Aware Thread Mapping 42
2.1.3 Parallel Applications and Sharing-Aware Data Mapping 48
2.1.4 Evaluation with a Microbenchmark 55
2.2 Related Work on Sharing-Aware Static Mapping 57
2.2.1 Static Thread Mapping 57
2.2.2 Static Data Mapping 58
2.2.3 Combined Static Thread and Data Mapping 59
2.3 Related Work on Sharing-Aware Online Mapping 59
2.3.1 Online Thread Mapping 59
2.3.2 Online Data Mapping 60
2.3.3 Combined Online Thread and Data Mapping 62
2.4 Discussion on Sharing-Aware Mapping and Related Work 62

3 LAPT – LOCALITY-AWARE PAGE TABLE 65
3.1 Structures Introduced by LAPT 65
3.2 Data Sharing Detection 66
3.3 Thread Mapping Computation 66

3.4 Data Mapping Computation . . . 67
3.5 Example of the Operation of LAPT . . . 68
3.6 Overhead . . . 69
3.6.1 Memory Storage Overhead . . . 69
3.6.2 Circuit Area Overhead . . . 69
3.6.3 Runtime Overhead . . . 70

4 SAMMU – SHARING-AWARE MEMORY MANAGEMENT UNIT . . . 71
4.1 Overview of SAMMU . . . 71
4.2 Structures Introduced by SAMMU . . . 72
4.3 Gathering Information about Memory Accesses . . . 72
4.4 Detecting the Sharing Pattern for Thread Mapping . . . 77
4.5 Detecting the Page Usage Pattern for Data Mapping . . . 78
4.6 Example of the Operation of SAMMU . . . 79
4.7 Implementation Details . . . 80
4.7.1 Hardware Implementation . . . 80
4.7.2 Notifying the Operating System About Page Migrations . . . 81
4.7.3 Performing the Thread Mapping . . . 81
4.7.4 Increasing the Supported Number of Threads . . . 81
4.7.5 Handling Context Switches and TLB Shootdowns . . . 82
4.8 Overhead . . . 82
4.8.1 Memory Storage Overhead . . . 82
4.8.2 Circuit Area Overhead . . . 82
4.8.3 Execution Time Overhead . . . 83

5 IPM – INTENSE PAGES MAPPING . . . 85
5.1 Detecting Memory Access Patterns via TLB Time . . . 85
5.2 Overview of IPM . . . 86
5.3 Structures Introduced by IPM . . . 87
5.4 Detailed Description of IPM . . . 88
5.4.1 Calculating the TLB Time . . . 88
5.4.2 Gathering Information for Thread Mapping . . . 88
5.4.3 Gathering Information for Data Mapping . . . 89
5.5 Overhead . . . 90
5.5.1 Memory Storage Overhead . . . 90
5.5.2 Circuit Area Overhead . . . 91
5.5.3 Execution Time Overhead . . . 91

6 EVALUATION OF OUR PROPOSALS 93
6.1 Methodology 93
6.1.1 Algorithm to Map Threads to Cores 93
6.1.2 Evaluation Inside a Full System Simulator 93
6.1.3 Evaluation Inside Real Machines with Software-Managed TLBs 95
6.1.4 Trace-Based evaluation Inside Real Machines with Hardware-Managed TLBs . 96
6.1.5 Presentation of the Results 97
6.1.6 Workloads 97
6.2 Comparison Between our Proposed Mechanisms 98
6.3 Accuracy of the Proposed Mechanisms 99
6.3.1 Accuracy of the Sharing Patterns for Thread Mapping 99
6.3.2 Accuracy of the Data Mapping 100
6.4 Performance Experiments 104
6.4.1 Performance Results in Simics/GEMS 104
6.4.2 Performance Results in Itanium 106
6.4.3 Performance Results in Xeon32 and Xeon64 108
6.4.4 Summary of Performance Results 109
6.5 Energy Consumption Experiments 110
6.6 Overhead 110
6.7 Design Space Exploration 113
6.7.1 Varying the Cache Memory Size (SAMMU) 113
6.7.2 Varying the Memory Page Size (SAMMU) 114
6.7.3 Varying the Interchip Interconnection Latency (SAMMU) 115
6.7.4 Varying the Multiplication Threshold (LAPT) 116
6.7.5 Analyzing the Impact of Aging (CR_{aging} in IPM) 117
6.7.6 Analyzing the Impact of the Migration Threshold (CR_{mig} in IPM) 118

7 CONCLUSIONS AND FUTURE WORK 119
7.1 Conclusions 119
7.2 Future Work 120
7.3 Publications 121

APPENDIX A THREAD MAPPING ALGORITHM 123
A.1 Related Work 124
A.2 EagerMap – Greedy Hierarchical Mapping 124
A.2.1 Description of the EagerMap Algorithm 126
A.2.2 Mapping Example in a Multi-level Hierarchy 130
A.2.3 Complexity of the Algorithm 132

A.3 Evaluation of EagerMap . . . 133
A.3.1 Methodology of the Mapping Comparison . . . 133
A.3.2 Results . . . 135
A.4 Conclusions . . . 138

APPENDIX B SUMMARY IN PORTUGUESE . . . 141
B.1 Introdução . . . 141
B.2 Visão Geral Sobre Mapeamento Baseado em Compartilhamento . . . 142
B.3 Mecanismos Propostos para Mapeamento Baseado em Compartilhamento . . . 144
B.3.1 LAPT – Locality-Aware Page Table . . . 144
B.3.2 SAMMU – Sharing-Aware Memory Management Unit . . . 144
B.3.3 IPM – Intense Pages Mapping . . . 145
B.4 Avaliação dos Mecanismos Propostos . . . 146
B.5 Conclusão e Trabalhos Futuros . . . 147

REFERENCES . . . 149

LIST OF ABBREVIATIONS AND ACRONYMS

AC	Access Counter
AT	Access Threshold
DBA	Dynamic Binary Analysis
FSB	Front Side Bus
IBS	Instruction-Based Sampling
ID	Identifier
ILP	Instruction Level Parallelism
IPC	Instructions per Cycle
ISA	Instruction Set Architecture
LRU	Least Recently Used
MRU	Most Recently Used
NUMA	Non-Uniform Memory Access
NV	NUMA Vector
PHT	Page History Table
PMU	Performance Monitoring Unit
PU	Processing Unit
QPI	QuickPath Interconnect
SM	Sharing Matrix
SMP	Symmetric Multi-Processor
SMT	Simultaneous Multithreading
SV	Sharers Vector
TAT	TLB Access Table
TLB	Translation Lookaside Buffer
TLP	Thread Level Parallelism
TSC	Time Stamp Counter
UMA	Uniform Memory Access

LIST OF FIGURES

1.1 Sandy Bridge architecture with two processors. Each processor consists of eight 2-way SMT cores and three cache levels. 24
1.2 Analysis of parallel applications, adapted from Diener et al. (2015b). . . . 25
1.3 Results obtained with sharing-aware mapping, adapted from Diener et al. (2014). 27
1.4 Results obtained with different metrics for sharing-aware mapping. 28
1.5 General behavior of the MMU with a hardware-managed TLB. 30
1.6 General behavior of the MMU with a software-managed TLB. 30

2.1 System architecture with two processors. Each processor consists of eight 2-way SMT cores and three cache levels. 36
2.2 The relation between cache coherence protocols and sharing-aware thread mapping. 37
2.3 How a good mapping can reduce the amount of invalidation misses in the caches. In this example, core 0 reads the data, then core 1 writes, and finally core 0 reads the data again. 37
2.4 Impact of sharing-aware mapping on replication misses. 38
2.5 System architecture with two processors. Each processor consists of eight 2-way SMT cores and three cache levels. 39
2.6 Harpertown architecture. 40
2.7 AMD Abu Dhabi architecture. 40
2.8 Intel Montecito/SGI NUMAlink architecture. 41
2.9 How the granularity of the sharing pattern affects the spatial false sharing problem. 43
2.10 Temporal false sharing problem. 44
2.11 Sharing patterns of parallel applications from the OpenMP NAS Parallel Benchmarks and the PARSEC Benchmark Suite. Axes represent thread IDs. Cells show the amount of accesses to shared data for each pair of threads. Darker cells indicate more accesses. The sharing patterns were generated using a memory block size of 4 KBytes, and storing 2 sharers per memory block. 45
2.12 Sharing patterns of CG varying the memory block size. 47
2.13 Sharing patterns of Facesim varying the memory block size. 47
2.14 Sharing patterns of SP varying the memory block size. 47
2.15 Sharing patterns of LU varying the memory block size. 47
2.16 Sharing patterns of Fluidanimate varying the memory block size. 47
2.17 Sharing patterns of Ferret varying the memory block size. 47

2.18 Sharing patterns of BT varying the number of sharers. 49
2.19 Sharing patterns of LU varying the number of sharers. 49
2.20 Sharing patterns of Fluidanimate varying the number of sharers. 49
2.21 Sharing patterns of Ferret varying the number of sharers. 49
2.22 Sharing patterns of Swaptions varying the number of sharers. 49
2.23 Sharing patterns of Blackscholes varying the number of sharers. 49
2.24 Influence of the page size on data mapping. 50
2.25 Influence of thread mapping on data mapping. In this example, threads 0 and 1 access page P. 51
2.26 Page exclusivity of several applications in relation to the total number of memory accesses. 52
2.27 Exclusivity of several applications. 53
2.28 Page exclusivity of several applications in relation to the total number of memory accesses varying the page size. 54
2.29 Page exclusivity of several applications in relation to the total number of memory accesses varying the thread mapping. 54
2.30 Producer-consumer microbenchmark results. 56

3.1 Overview of the MMU and LAPT. 65
3.2 Detailed description of the LAPT mechanism. In this example, thread 3 generates a TLB miss in page P. 68

4.1 Overview of the MMU and SAMMU. 71
4.2 Value of AT_{new} relative to AT_{old}. For Equation 4.42, we consider AT_{old} as $10M$. 76
4.3 Behavior of Equations 4.23 and 4.42 varying AT_{old} from $100K$ to $10M$. . . 76
4.4 Operation of SAMMU. Structures that were added to the normal operation of the hardware and operating system are marked in **bold**. The system consists of 4 NUMA nodes with 2 cores each. The NUMA Threshold is 4. Page P is initially located on NUMA node 0. In this example, thread 3 (core 3, NUMA node 1) saturates the AC of page P. 78

5.1 Accuracy results obtained with different metrics for sharing-aware mapping. 85
5.2 Overview of the MMU and IPM. 87

6.1 Sharing patterns of BT. 101
6.2 Sharing patterns of CG. 101
6.3 Sharing patterns of DC. 101
6.4 Sharing patterns of EP. 101
6.5 Sharing patterns of FT. 101
6.6 Sharing patterns of IS. 101

6.7 Sharing patterns of LU. 101
6.8 Sharing patterns of MG. 101
6.9 Sharing patterns of SP. 101
6.10 Sharing patterns of UA. 101
6.11 Sharing patterns of Blackscholes. 101
6.12 Sharing patterns of Bodytrack. 101
6.13 Sharing patterns of Canneal. 102
6.14 Sharing patterns of Dedup. 102
6.15 Sharing patterns of Facesim. 102
6.16 Sharing patterns of Ferret. 102
6.17 Sharing patterns of Fluidanimate. 102
6.18 Sharing patterns of Freqmine. 102
6.19 Sharing patterns of Raytrace. 102
6.20 Sharing patterns of Swaptions. 102
6.21 Sharing patterns of Streamcluster. 102
6.22 Sharing patterns of Vips. 102
6.23 Sharing patterns of x264. 102
6.24 Accuracy of the data mappings using our proposed mechanisms and the first-touch policy (default policy of Linux). The gray line over the bars represents the exclusivity level of the applications (Section 2.1.3.2). 103
6.25 Results in Simics/GEMS normalized to the default mapping (Equation 6.1). 105
6.26 Execution time in Itanium normalized to the operating system mapping (Equation 6.1). Lower results are better. 107
6.27 Performance results in Xeon32 normalized to the operating system mapping (Equation 6.1). Lower results are better. 109
6.28 Performance results in Xeon64 normalized to the operating system mapping (Equation 6.1). Lower results are better. 110
6.29 Energy consumption in Xeon32 normalized to the operating system mapping (Equation 6.1). Lower results are better. 111
6.30 Performance overhead on the hardware and software levels. Lower results are better. 112
6.31 Varying L2 cache memory sizes in Simics/GEMS with SAMMU. Results are normalized to the default mapping with the corresponding cache size. Lower results are better. 114
6.32 Varying memory page sizes in Simics/GEMS with SAMMU. Results of Figures 6.32a and 6.32b are normalized to the default mapping with the corresponding page size. In Figures 6.32a and 6.32b, lower results are better. 115
6.33 Varying interchip interconnection latency in Simics/GEMS with SAMMU. Lower results are better. 116

6.34 Varying the multiplication threshold. Figure 6.34a is normalized to the default mapping, where lower results are better. 117
6.35 Normalized execution time varying the aging in IPM running in Itanium. The higher CR_{aging}, less aging. Lower results are better. 117
6.36 Normalized execution time varying the migration threshold in IPM running in Itanium. The higher CR_{mig}, less migrations. Lower results are better. . . 118

A.1 Example communication matrices for parallel applications consisting of 64 tasks. Axes represent task IDs. Cells show the amount of communication between tasks. Darker cells indicate more communication. 125
A.2 Mapping 16 tasks in an architecture with 8 PUs, L3 caches shared by 2 PUs, 2 processors per machine, and 2 machines. 128
A.3 Communication matrices used in our example. 131
A.4 Example of the $GenerateGroup$ algorithm, when generating the first group in our example. 131
A.5 Hardware architecture used in our experiments. 134
A.6 Execution time (in ms) of the mapping algorithms, for different numbers of tasks to be mapped. Lower results are better. 135
A.7 Comparison of the mapping quality, normalized to the random mapping. Higher values are better. 136
A.8 Comparison of the mapping stability. Number of migrations depending on the percentile change of the communication matrix for an application consisting of 64 tasks. Less migrations are better. 137
A.9 Performance improvement compared to the OS mapping of the NAS-MPI and NAS-OMP benchmarks. Higher results are better. 138
A.10 Execution time of HPCC using online mapping. Lower results are better. . 138

B.1 Visão geral do comportamento da MMU e do LAPT. 144
B.2 Visão geral da MMU e do SAMMU. 145
B.3 Visão geral da MMU e do IPM. 146

LIST OF TABLES

2.1 Summary of related work. 63

6.1 Configuration of the experiments. 94

ABSTRACT

As thread-level parallelism increases in modern architectures due to larger numbers of cores per chip and chips per system, the complexity of their memory hierarchies also increase. Such memory hierarchies include several private or shared cache levels, and Non-Uniform Memory Access nodes with different access times. One important challenge for these architectures is the data movement between cores, caches, and main memory banks, which occurs when a core performs a memory transaction. In this context, the reduction of data movement is an important goal for future architectures to keep performance scaling and to decrease energy consumption. One of the solutions to reduce data movement is to improve memory access locality through *sharing-aware thread and data mapping*.

State-of-the-art mapping mechanisms try to increase locality by keeping threads that share a high volume of data close together in the memory hierarchy (*sharing-aware thread mapping*), and by mapping data close to where its accessing threads reside (*sharing-aware data mapping*). Many approaches focus on either thread mapping or data mapping, but perform them separately only, losing opportunities to improve performance. Some mechanisms rely on execution traces to perform a static mapping, which have a high overhead and can not be used if the behavior of the application changes between executions. Other approaches use sampling or indirect information about the memory access pattern, resulting in imprecise memory access information.

In this thesis, we propose novel solutions to identify an optimized sharing-aware mapping that make use of the memory management unit of processors to monitor the memory accesses. Our solutions work online in parallel to the execution of the application and detect the memory access pattern for both thread and data mappings. With this information, the operating system can perform sharing-aware thread and data mapping during the execution of the application, without any prior knowledge of their behavior. Since they work directly in the memory management unit, our solutions are able to track most memory accesses performed by the parallel application, with a very low overhead. They can be implemented in architectures with hardware-managed TLBs with little additional hardware, and some can be implemented in architectures with software-managed TLBs without any hardware changes. Our solutions have a higher accuracy than previous mechanisms because they have access to more accurate information about the memory access behavior.

To demonstrate the benefits of our proposed solutions, we evaluate them with a wide variety of applications using a full system simulator, a real machine with software-managed TLBs, and a trace-driven evaluation in two real machines with hardware-managed TLBs. In the experimental evaluation, our proposals were able to reduce execution time by up to 39%. The improvements happened to a substantial reduction in cache misses and interchip interconnection traffic.

Keywords: Data movement. Thread and data mapping. Cache memory. NUMA.

Mapeamento Dinâmico de Threads e Dados Usando a Unidade de Gerência de Memória

RESUMO

Conforme o paralelismo a nível de *threads* aumenta nas arquiteturas modernas devido ao aumento do número de núcleos por processador e processadores por sistema, a complexidade da hierarquia de memória também aumenta. Tais hierarquias incluem diversos níveis de *caches* privadas ou compartilhadas e tempo de acesso não uniforme à memória. Um desafio importante em tais arquiteturas é a movimentação de dados entre os núcleos, *caches* e bancos de memória primária, que ocorre quando um núcleo realiza uma transação de memória. Neste contexto, a redução da movimentação de dados é um dos pilares para futuras arquiteturas para manter o aumento de desempenho e diminuir o consumo de energia. Uma das soluções adotadas para reduzir a movimentação de dados é aumentar a localidade dos acessos à memória através do mapeamento de *threads* e dados.

Mecanismos de mapeamento do estado-da-arte aumentam a localidade de memória mapeando *threads* que compartilham um grande volume de dados em núcleos próximos na hierarquia de memória (mapeamento de *threads*), e mapeando os dados em bancos de memória próximos das *threads* que os acessam (mapeamento de dados). Muitas propostas focam em mapeamento de *threads* ou dados separadamente, perdendo oportunidades de ganhar desempenho. Outras propostas dependem de traços de execução para realizar um mapeamento estático, que podem impor uma sobrecarga alta e não podem ser usados em aplicações cujos comportamentos de acesso à memória mudam em diferentes execuções. Há ainda propostas que usam amostragem ou informações indiretas sobre o padrão de acesso à memória, resultando em informação imprecisa sobre o acesso à memória.

Nesta tese de doutorado, são propostas soluções inovadoras para identificar um mapeamento que otimize o acesso à memória fazendo uso da unidade de gerência de memória para monitor os acessos à memória. As soluções funcionam dinamicamente em paralelo com a execução da aplicação, detectando informações para o mapeamento de *threads* e dados. Com tais informações, o sistema operacional pode realizar o mapeamento durante a execução das aplicações, não necessitando de conhecimento prévio sobre o comportamento da aplicação. Como as soluções funcionam diretamente na unidade de gerência de memória, elas podem monitorar a maioria dos acessos à memória com uma baixa sobrecarga. Em arquiteturas com TLB gerida por *hardware*, as soluções podem ser implementadas com pouco *hardware* adicional. Em arquiteturas com TLB gerida por *software*, algumas das soluções podem ser implementadas sem *hardware* adicional. As soluções aqui propostas possuem maior precisão que outros mecanismos porque possuem acesso a mais informações sobre o acesso à memória.

Para demonstrar os benefícios das soluções propostas, elas são avaliadas com uma variedade de aplicações usando um simulador de sistema completo, uma máquina real com TLB gerida por *software*, e duas máquinas reais com TLB gerida por *hardware*. Na avaliação experimental, as

soluções reduziram o tempo de execução em até 39%. O ganho de desempenho se deu por uma redução substancial da quantidade de faltas na *cache*, e redução do tráfego entre processadores.

Palavras-chave: Movimentação de dados, Mapeamento de threads e dados, Memória cache, NUMA.

1 INTRODUCTION

Since the beginning of the information era, the demand for computing power has been unstoppable. Whenever the technology advances enough to fulfill the needs of a time, new and more complex problems arise, such that the technology is again insufficient to solve them. In the past, the increase of the performance happened mainly due to instruction level parallelism (ILP), with the introduction of several pipeline stages, out-of-order and speculative execution. The increase of the clock rate frequency was also a important way to improve performance. However, the available ILP exploited by compilers and architectures is reaching its limits (CABEZAS; STANLEY-MARBELL, 2011). The increase of clock frequency is also reaching its limits because it raises the energy consumption, which is an important issue for current and future architectures (TOLENTINO; CAMERON, 2012).

To keep performance increasing, processor architectures are becoming more dependent on thread level parallelism (TLP), employing several cores to compute in parallel. These parallel architectures put more pressure on the memory subsystem, since more bandwidth to move data between the cores and the main memory is required. To handle the additional bandwidth, current architectures introduce complex memory hierarchies, formed by multiple cache levels, some composed by multiple banks connected to a memory controller. The memory controller interfaces a Uniform or Non-Uniform Memory Access (UMA or NUMA) system. However, with the upcoming increase of the number of cores, a demand for an even higher memory bandwidth is expected (COTEUS et al., 2011).

In this context, the reduction of data movement is an important goal for future architectures to keep performance scaling and to decrease energy consumption (BORKAR; CHIEN, 2011). Most of data movement in current architectures occur due to memory accesses and *communication* between the threads. The communication itself in shared memory environments is performed through *accesses to blocks of memory shared between the threads*. One of the solutions to reduce data movement consists of improving the memory locality (TORRELLAS, 2009). In this work, we improve memory locality by performing a global scheduling (CASAVANT; KUHL, 1988) of threads and data of parallel applications considering their memory access behavior. We perform the global scheduling by setting a mapping function in the scheduler. In the thread mapping, threads that share data are mapped to cores that are close to each other. In the data mapping, the data is mapped close to the cores that are accessing it. This type of thread and data mapping is called *sharing-aware* mapping.

In the rest of this chapter, we first describe how sharing-aware mapping improves memory locality and thereby the performance and energy efficiency. Afterwards, we explain the challenges of detecting the necessary information to perform sharing-aware mapping and the benefits of a hardware based approach. Then, we introduce the contributions of this thesis. Finally, we show how we organized the text.

1.1 Improving Memory Locality With Sharing-Aware Mapping

During the execution of a multi-threaded application, the mapping of its threads and their data can have a great impact on the memory hierarchy, both in performance and energy consumption (FELIU et al., 2012). The potential for improvements depends on the architecture of the machines, as well as on the memory access behavior of the parallel application. Considering the architecture, each processor family uses a different organization of the cache hierarchy and the main memory system, such that the mapping impact varies among different systems. In general, the cache hierarchy is formed by multiple levels, where the levels closer to the processor cores tend to be private, followed by caches shared by multiple cores. For NUMA systems, besides the cache hierarchy, the main memory is also clustered between cores or processors. These complex memory hierarchies introduce differences in the memory access latencies and bandwidths and thereby in the memory locality, which vary depending on the core that requested the memory operation, the target memory bank and, if the data is cached, which cache resolved the operation.

An example of architecture that is affected by sharing-aware thread and data mapping is the Intel Sandy Bridge architecture (INTEL, 2012a), illustrated in Figure 1.1. Sandy Bridge is a NUMA architecture, as we can observe that each processor is connected to a local memory bank. An access to a remote memory bank, a remote access, has an extra performance penalty because of the interchip interconnection latency. Virtual cores within a core can share data using the L1 and L2 cache levels, and all cores within a processor can share data with the L3 cache. Data that is accessed by cores from different caches require the cache coherence protocol to keep the consistency of the data due to write operations.

Regarding the parallel applications, the main challenge of sharing-aware mapping in shared memory based parallel applications is to detect data sharing. This happens because, in these applications, the source code does not express explicitly which memory accesses happen to a block of memory accessed by more than one thread. This data sharing then is considered as an *implicit* communication between the threads, and occurs whenever a thread access a block of memory that is also accessed by other thread. Programming environments include Pthreads and

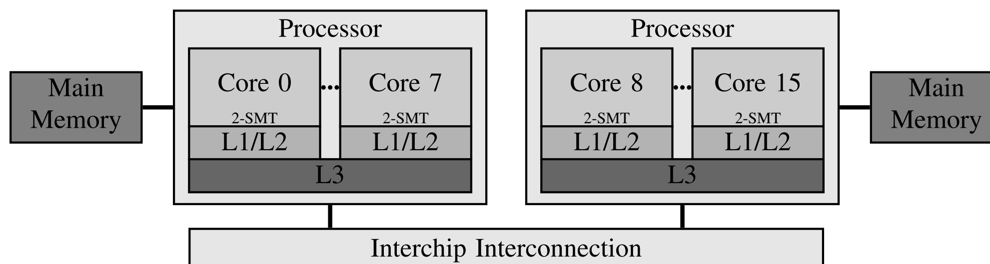
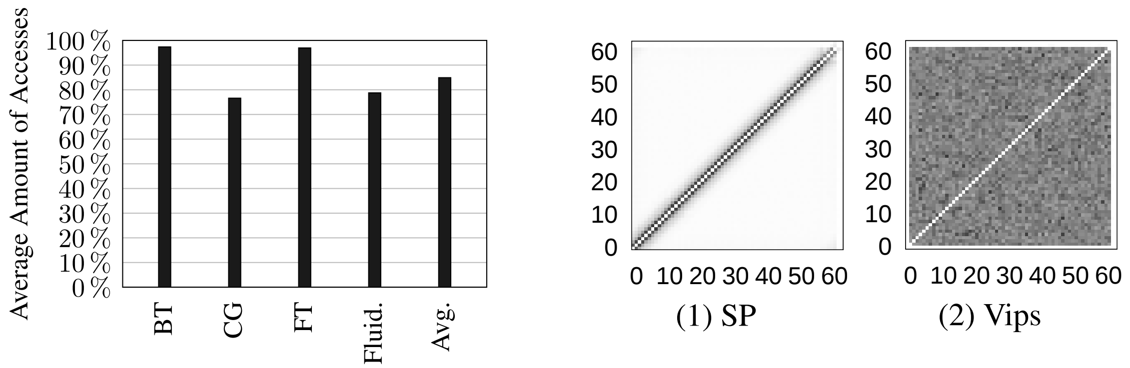


Figure 1.1: Sandy Bridge architecture with two processors. Each processor consists of eight 2-way SMT cores and three cache levels.



(a) Average amount of memory accesses to a given page performed from a single NUMA node. The highest the value, more potential for data mapping.

(b) Example sharing patterns of applications. Axes represent thread IDs. Cells show the amount of accesses to shared pages for each pair of threads. Darker cells indicate more accesses.

Figure 1.2: Analysis of parallel applications, adapted from Diener et al. (2015b).

OpenMP (OPENMP, 2013). On the other hand, there are parallel applications that are based on message passing, where threads send messages to each other using specific routines provided by a message passing library, such as MPI (GABRIEL et al., 2004). In such applications, the communication is *explicit* and can be monitored by tracking the messages sent. Due to the implicit communication, parallel applications using shared memory present more challenges for mapping and are the focus of this thesis.

In the context of shared memory based parallel applications, the memory access behavior influences sharing-aware mapping because threads can share data among themselves differently for each application. In some applications, each thread has its own data set, and very little memory is shared between threads. On the other hand, threads can share most of their data, imposing a higher overhead on cache coherence protocols to keep consistency among the caches. In applications that have a lot of shared memory, the way the memory is shared is also important to be considered. Threads can share a similar amount of data with all the other threads, or can share more data within a subgroup of threads. In general, sharing-aware data mapping is able to improve performance in all applications except when all data of the application is equally shared between the threads, while sharing-aware thread mapping improves performance in applications whose threads share more data within a subgroup of threads.

In Diener et al. (2015b), it is shown that, on average, 84.9% of the memory accesses to a given page are performed from a single NUMA node, considering a 4 NUMA node machine and 4 KBytes memory pages. Results regarding this potential for data mapping is found in Figure 1.2a. This result indicates that most applications have a high potential for sharing-aware data mapping, since, on average, 84.9% of the whole of the memory accesses could be improved by having a more efficient mapping of pages to NUMA nodes. Nevertheless, the potential is different between applications, in which we can observe in the applications shown

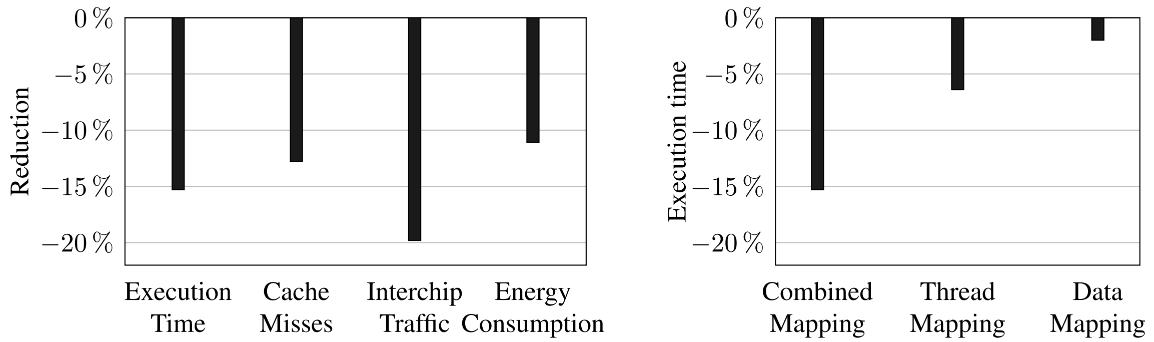
in Figure 1.2a, where the values range from 76.6%, in CG, to 97.4%, in BT. These different behaviors between applications impose a challenge to online mapping mechanisms, as they need to adapt to the behavior automatically during runtime.

Regarding the mapping of threads to cores, Diener et al. (2015b) also shows that parallel applications can present a wide variety of data sharing patterns. Several patterns that can be exploited by sharing-aware thread mapping were found in the applications. For instance, in Figure 1.2b1, neighbor threads share data, which is common in applications parallelized using domain decomposition. Other patterns suitable for mapping include patterns in which distant threads share data, or even patterns in which threads share data in clusters due to a pipeline parallelization model. On the other hand, applications such as Vips, whose sharing pattern is found in Figure 1.2b2, each thread has a similar amount of data shared to all threads, such that no thread mapping can optimize the communication. Nevertheless, in these applications, data mapping still can improve performance by optimizing the memory accesses to the data private to each thread. The analysis reveal that most parallel application can benefit from sharing-aware mapping.

Sharing-aware thread and data mapping improves performance and energy efficiency of parallel applications by optimizing memory accesses (DIENER et al., 2014). Improvements happen for three main reasons. First, cache misses are reduced by decreasing the number of invalidations that happen when write operations are performed on shared data (MARTIN; HILL; SORIN, 2012). For read operations, the effective cache size is increased by reducing the replication of cache lines on multiple caches (CHISHTI; POWELL; VIJAYKUMAR, 2005). Second, the locality of memory accesses is increased by mapping data to the NUMA node where it is most accessed. Third, the usage of interconnections in the system is improved by reducing the traffic on slow and power-hungry interchip interconnections, using more efficient intrachip interconnections instead. Although there are several benefits of using sharing-aware thread and data mapping, the wide variety of architectures, memory hierarchies and memory access behavior of parallel applications restricts its usage in current systems.

The performance and energy consumption improvements that can be obtained by sharing-aware mapping are evaluated in Diener et al. (2014). The results using an *Oracle* mapping, which generates a mapping considering all memory accesses performed by the application, is shown in Figure 1.3a. Experiments were performed in a machine with 4 NUMA nodes capable of running 64 threads simultaneously. The average execution time reduction reported was 15.3%. The reduction of execution time was possible due to a reduction of 12.8% of cache misses, and 19.8% of interchip interconnection traffic. Results were highly dependent on the characteristics of the parallel application. In applications that exhibit a high potential for sharing-aware mapping, execution time was reduced by up to 39.5%. Energy consumption was also reduced, by 11.1% on average.

Another interesting result is that thread and data mapping should be performed together to achieve higher improvements (DIENER et al., 2014). Figure 1.3b shows the average execution



(a) Average reduction of execution time, L3 cache misses, interchip interconnection traffic and energy consumption when using a combined thread and data mappings.

(b) Average execution time reduction provided by a combined thread and data mapping, and by using thread and data mapping separately.

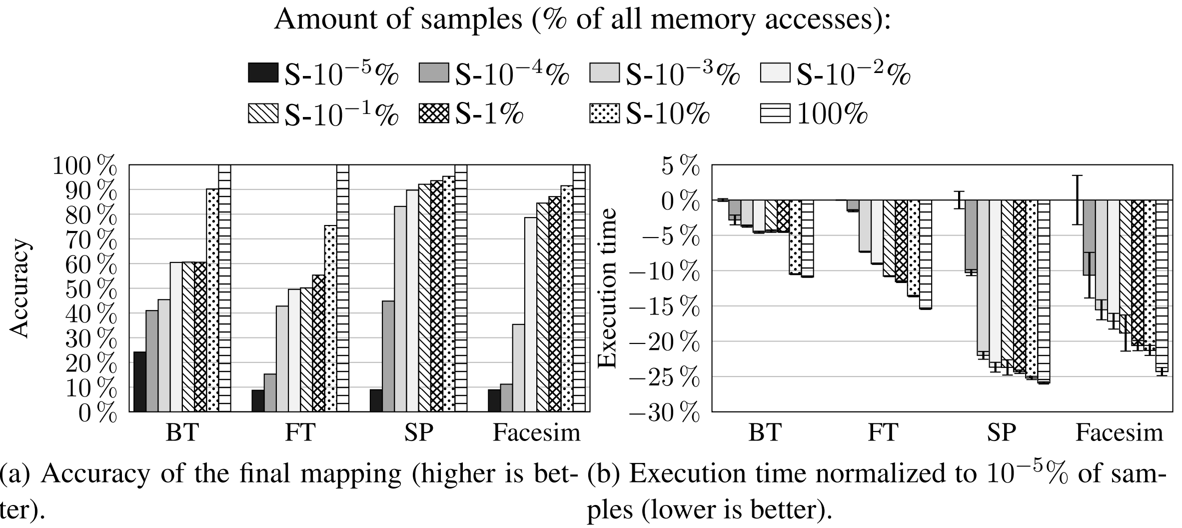
Figure 1.3: Results obtained with sharing-aware mapping, adapted from Diener et al. (2014).

time reduction of using thread and data mapping together, and using them separately. The execution time reduction of using a combined thread and data mapping was 15.3%. On the other hand, the reduction of using thread mapping alone was 6.4%, and using data mapping alone 2.0%. It is important to note that the execution time reduction of a combined mapping is higher than the sum of the reductions of using the mappings separately. This happens because each mapping has positive effect in the other mapping, which is discussed further in this thesis in more details.

1.2 Monitoring Memory Accesses for Sharing-Aware Mapping

The main information required to perform sharing-aware mapping are the memory addresses accessed by each thread. This can be done statically or online. When memory addresses are monitored statically, the application is usually executed in controlled environments such as simulators, where all memory addresses can be tracked (BARROW-WILLIAMS; FENSCH; MOORE, 2009). These techniques are not able to handle applications whose sharing pattern changes between execution. Also, the amount of different memory hierarchies present in current and future architectures limit the applicability of static mapping techniques. In online mapping, the sharing information must be detected while running the application. As online mapping mechanisms require memory access information to infer memory access patterns and make decisions, different information collection approaches have been employed with varying degrees of accuracy and overhead. Although capturing all memory accesses from an application would provide the best information for mapping algorithms, the overhead would surpass the benefits from better task and data mappings. For this reason, this is only done for static mapping mechanisms.

In order to achieve a smaller overhead, most traditional methods for collecting memory



(a) Accuracy of the final mapping (higher is better).

(b) Execution time normalized to 10^{-5}% of samples (lower is better).

Figure 1.4: Results obtained with different metrics for sharing-aware mapping.

access information are based on sampling. Memory access patterns can be estimated by tracking page faults (DIENER et al., 2014; DIENER et al., 2015a; LAROWE; HOLLIDAY; ELLIS, 1992; CORBET, 2012a; CORBET, 2012b), cache misses (AZIMI et al., 2009) or TLB misses (MARATHE; THAKKAR; MUELLER, 2010; VERGHESE et al., 1996), or by using hardware performance counters (DASHTI et al., 2013), among others. Still, these sampling based mechanisms present an accuracy lower than intended, as we show in the next paragraphs. This is due to the small number of memory access captured (and their representativeness) in relation to all of the memory accesses. For instance, a thread may wrongly appear to access a page less than another thread because its memory accesses were undersampled. In another scenario, a thread may have few access to a page, while having cache or TLB misses in most of these accesses, leading to bad mappings in mechanisms based on cache or TLB misses.

Following the hypothesis that a more accurate estimation of the memory access pattern of an application can result in a better mapping and performance improvements, we performed experiments varying the amount of memory samples used to calculate the mapping. Experiments were run using the NAS parallel benchmarks (JIN; FRUMKIN; YAN, 1999) and the PARSEC benchmark suite (BIENIA et al., 2008) on a two NUMA node machine (more information in Section 6.1.3). Memory accesses were sampled using Pin (BACH et al., 2010). We vary the number of samples from 10^{-5}% to using all memory accesses. A realistic sampling based mechanism (DIENER et al., 2015a) uses at most 10^{-3}% of samples to avoid harming performance. Another sampling based mechanism (DASHTI et al., 2013) specify that it samples once every 130,000 cycles.

The accuracy and execution time obtained with the different methods for benchmarks BT, FT, SP, and Facesim are presented in Figure 1.4. To calculate the accuracy, we compare if the NUMA node selected for each page of the applications is equal to the NUMA node that

performed most accesses to the page (the higher the percentage, the better). The execution time is calculated as the reduction compared to using $10^{-5}\%$ samples (the bigger the reduction, the better). The accuracy results, in Figure 1.4a, show that accuracy increases with the number of memory access samples used, as expected. It shows that some applications require more samples than others. For instance, BT and FT require at least 10% of samples to achieve a good accuracy, while SP and Facesim achieve a good accuracy using $10^{-2}\%$ of samples.

The execution time results, in Figure 1.4b, indicate a clear trend: a higher accuracy in estimating memory access patterns translates to a shorter execution time. However, most sampling mechanisms are limited to a small number of samples due to the high overhead of the techniques used for capturing memory accesses (AZIMI et al., 2009; DIENER et al., 2014), harming accuracy and thereby performance. Given these considerations, the goal of this thesis is to propose mechanisms for online sharing-aware mapping that have access to a high amount of memory addresses to maximize the accuracy of the detected pattern, while keeping the overhead low. The mechanism should detect the information without the need for modifying any source code, and provide it to the operating system to perform the mapping, optimizing performance and energy consumption.

To achieve these goals, we include hardware support in the architectures. In this hybrid approach, the hardware must provide information about the memory access, and the software uses the information to perform sharing-aware mapping. The hardware support must impose a low overhead, considering hardware cost, performance and overhead. The memory management unit (MMU) present in most architectures can be used to analyze the memory access pattern of parallel applications, which is the subject of the next section.

1.3 Using the Memory Management Unit to Gather Information on Memory Accesses

Computer systems that support virtual memory use a MMU to translate virtual to physical addresses. Virtual memory provides security, avoiding that an application access memory addresses of other applications, and allows an application to access more memory than the amount of physical memory available in the machine. It also provides a transparent management of the entire memory address space, removing this burden from the application. To perform the address translation, the operating system stores page tables in the main memory, which contain the physical address and metadata of each memory page. The metadata is composed mostly by protection information, such as if the page is writable or contains binary code to be executed. In most operating systems, threads of a parallel application share the same page table, since all threads access the same memory address space. A special cache memory, the Translation Lookaside Buffer (TLB), is used to speed up the address translation. Depending on the architecture, the TLB can be managed by the hardware or by the software (TANENBAUM, 2007).

A high-level overview of the operation of the MMU using a *hardware-managed* TLB is illustrated in Figure 1.5. On every memory access, the MMU checks if the corresponding page

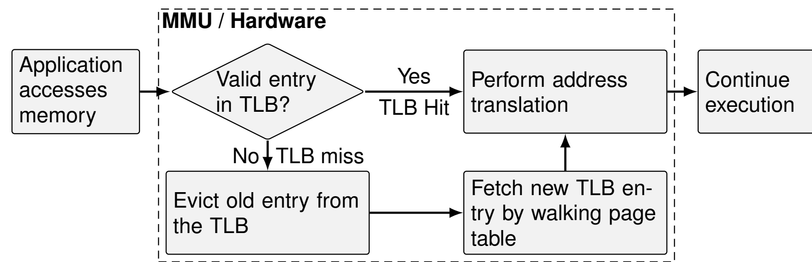


Figure 1.5: General behavior of the MMU with a hardware-managed TLB.

has a valid entry in the TLB. If it does, the virtual address is translated to a physical address and the memory access is performed. If the entry is not in the TLB, the MMU reads the entry from the page table in the main memory. This procedure is called page table walk. Most architectures have multi-level page tables to reduce storage space wasted when an application has low memory usage, or due to memory fragmentation. In multi-level page tables, the page table walk require several memory accesses, increasing the importance of the TLB. After performing the page table walk, the MMU caches the entry in the TLB and proceeds with the address translation and memory access. The main benefit of using an MMU with hardware-managed TLB is that it has a high performance.

The behavior of a *software-managed* TLB is a little different, and it is shown in Figure 1.6. When the application requests a memory access, the MMU checks if the corresponding page has its entry cached in the TLB. If it does, the behavior is the same as in the hardware-managed TLB: the hardware translates the virtual address to a physical address and then performs the memory access. However, in case there is no valid entry in the TLB, there is no hardware page table walker available, then the hardware generates an interrupt to the operating system. Upon receiving the interrupt, the operating system, in its TLB miss handler routine, walks through the page table in software, and inserts the entry in the TLB using instructions provided by the instruction set architecture (ISA). Afterwards, the system returns from the interrupt, the memory address is translated and the application continues its execution. Usually, the TLB miss

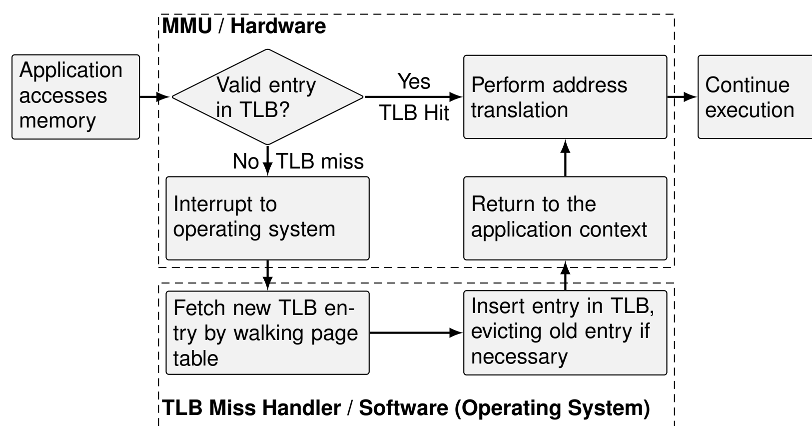


Figure 1.6: General behavior of the MMU with a software-managed TLB.

handler routine is written directly in Assembly code for maximum performance. Depending on the architecture, the operating system has access only to the TLB miss address, such as in Itanium (INTEL, 2010a), while in others it also has access to the contents of the TLB, such as in MIPS (MIPS, 1996). The main benefits of using a software-managed TLB is its simpler hardware, and that it allows the operating system to decide how to organize the page table. Some architectures, as Itanium, have a hybrid MMU, which can be configured to behave as a hardware-managed or software-managed TLB using control registers.

As we can observe, all memory accesses are handled by the MMU, whether it results in a TLB hit or miss. The MMU already handle memory addresses in a page level granularity, which is ideal to achieve our goal, since we focus on NUMA architectures. Also, each core has its own MMU and, in multi-threaded cores, each virtual core has its own virtual MMU. In this way, it is easy for the MMU to identify which thread of the parallel application is performing the memory accesses. Therefore, the MMU already provides most of the infrastructure required to analyze memory access pattern, which is also corroborated by some preliminary results (CRUZ; DIENER; NAVAUX, 2012; CRUZ; DIENER; NAVAUX, 2015). Due to these reasons, we chose to implement our proposal to provide support for sharing-aware mapping in the MMU. In case of a hardware-managed TLB, we add extra hardware to the MMU. In case of a software-managed TLB, we can implement most of our proposals in the TLB miss handler of the operating system.

1.4 Contributions of this Thesis

In this work, we propose three mechanisms to gather information on sharing-aware mapping. Our mechanisms focus on online mapping of shared memory based applications. The sharing detection is implemented directly in the memory management unit (MMU) of current processors, making use of their virtual memory implementation. We chose to implement directly into the MMU because it has access to all memory accesses. The thread and data mappings are then performed in software by the operating system using the information detected by the hardware. The proposed mechanisms to be implemented in the MMU are:

Locality-Aware Page Table (LAPT): LAPT extends the page table with special fields that allows operating systems to perform thread and data mapping in parallel applications. It keeps track of all Translation Lookaside Buffer (TLB) misses, detecting the sharing between the threads and registering a list of the latest threads that accessed each page in the corresponding page table entry. (CRUZ et al., 2014b)

Sharing-Aware Memory Management Unit (SAMMU): Similarly to LAPT, SAMMU extends the page table with information for thread and data mapping. However, SAMMU is able to consider all memory accesses to each page by adding an access counter to each TLB entry. Also, SAMMU already detects in hardware the most suitable NUMA node

for each page, notifying the operating system whenever a page migration should occur.

Intense Pages Mapping (IPM): IPM, like SAMMU, detects information for thread and data mappings directly in hardware. However, IPM has several advantages over SAMMU. The main advantage is that it has a much lower implementation cost, while keeping a similar accuracy. Also, it can be implemented natively in architectures with software-managed TLB.

Our proposals have the following features:

- They operate during the execution of the application, allowing the mapping to be performed online, without needing previous information about the application.
- By detecting the locality in hardware, we have access to many more memory access samples than previous mechanisms, generating more accurate information. More specifically, LAPT can track all memory accesses that result in TLB misses, while SAMMU and IPM are able to indirectly consider all memory accesses.
- Any shared memory based parallel application is able to benefit from the optimized mapping without modification to its source code.
- They do not depend on any particular parallelization API, since the mechanisms are implemented in the hardware and operating system.

To the best of our knowledge, our proposals are the first mechanisms that use the MMU to monitor the memory accesses and perform both online thread and data mapping. Also, SAMMU and IPM are the first mechanisms that detect the memory access pattern for thread and data mapping completely on the hardware level, considering all memory accesses to achieve maximum accuracy.

In this work, we also propose, as a secondary contribution, a thread mapping algorithm called **EagerMap** (CRUZ et al., 2015). EagerMap is an efficient algorithm to generate sharing-aware task mappings. It is designed to work with symmetric tree hierarchies, focusing on shared memory environments. It coarsens the application graph by grouping threads that share a lot of data. This coarsening follows the topology of the architecture hierarchy. Based on observations of application behavior, we propose an efficient greedy strategy to generate each group. It achieves a high accuracy and is faster than other current approaches. After the coarsening, we map the group graph to the architecture graph.

1.5 Organization of the Text

This thesis is organized as follows. Chapter 2 explains how sharing-aware mapping affects the performance and presents the related work. Chapter 3 describes the LAPT mechanism. Chapter 4 describes the SAMMU mechanism. Chapter 5 describes the IPM mechanism. Chapter 6 shows the methodology we adopted to evaluate our proposals and the results. Chapters 7

draws our conclusions and future work. Appendix A presents the EagerMap algorithm. Appendix B presents a summary of this thesis in the portuguese language, as required by the PPGC Graduate Program in Computing.

2 SHARING-AWARE MAPPING: OVERVIEW AND RELATED WORK

In this chapter, we present an overview of sharing-aware mapping and then describe related work. In the related work, we begin with a discussion on static mapping mechanisms, in which the mapping of an application is determined before its execution and do not change during the execution. Afterwards, we evaluate online mapping mechanisms, where the detection of the information required to map an application, as well as the mapping itself, must be performed during its execution. Both thread and data mapping mechanisms are covered. Finally, we describe a summary of current mapping strategies, contextualizing our proposals in the state-of-art.

2.1 Improving Memory Locality With Sharing-Aware Mapping

The memory locality present in parallel applications depends on their memory access behavior. More specifically, it depends on how the data is shared between the threads. A block of data is private if it is only accessed by one thread, otherwise, it is shared. The amount of memory accesses to data private to a thread, or data that are shared between several threads, influence both thread and data sharing-aware mappings. The potential for improvements using sharing-aware mapping depends on the architecture of the machines, as well as on the memory access behavior of the parallel application. We first explain how sharing-aware mapping influences memory locality in different types of architectures. Afterwards, we show how parallel applications are affected by it. Finally, we present a simple parallel application, a producer-consumer microbenchmark, and perform some experiments with it in a machine, demonstrating the potential of sharing-aware mapping.

2.1.1 Memory Locality in Different Architectures

The difference in the memory locality between the cores affects the data sharing performance. As parallel applications need to access shared data, complex memory hierarchy presents challenges for mapping threads to cores, and data to NUMA nodes (WANG et al., 2012). Threads that access a large amount of shared data should be mapped to cores that are close to each other in the memory hierarchy, while data should be mapped to the NUMA node executing the threads that access them (RIBEIRO et al., 2009). In this way, the *locality* of the memory accesses is improved, which leads to an increase of performance and energy efficiency. For optimal performance improvements, data and thread mapping should be performed together (TERBOVEN et al., 2008).

In this section, we first analyze the theoretical benefits that an improved memory locality provide to shared memory architectures. Afterwards, we present examples of current architectures, and how memory locality impacts in their performance. Finally, we briefly describe how

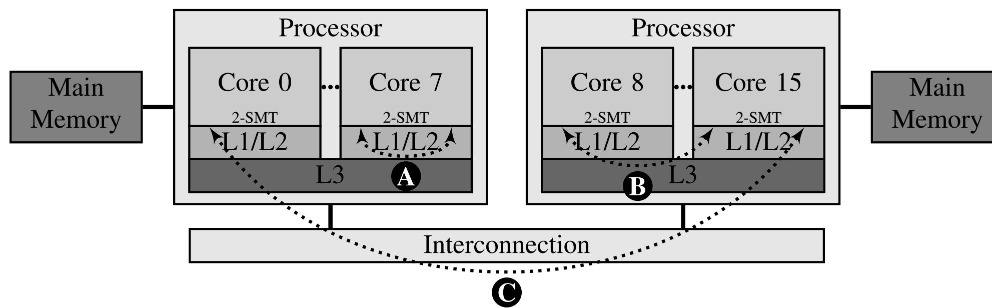


Figure 2.1: System architecture with two processors. Each processor consists of eight 2-way SMT cores and three cache levels.

locality affects the performance in network clusters and grids.

2.1.1.1 Understanding Memory Locality in Shared Memory Architectures

A sharing-aware thread and data mapping is able to improve the memory locality in shared memory architectures, and thereby the performance and energy efficiency, due to these main reasons (the reasons are very related to each other):

Lower Latency when Sharing Data: To illustrate how the memory hierarchy influences memory locality, Figure 2.1 shows an example of an architecture where there are three possibilities for sharing between threads. Threads running on the same core (Figure 2.1 Ⓐ) can share data through the fast L1 or L2 caches and have the highest sharing performance. Threads that run on different cores (Figure 2.1 Ⓑ) have to share data through the slower L3 cache, but can still benefit from the fast intra-chip interconnection. When threads share data across physical processors in different NUMA nodes (Figure 2.1 Ⓒ), they need to use the slow inter-chip interconnection. Hence, the sharing performance in case Ⓒ is the slowest in this architecture.

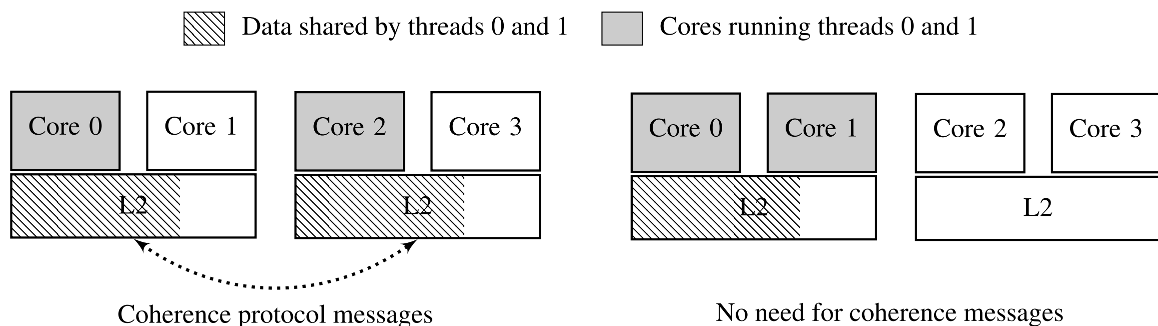
Reducing the impact of cache coherence protocols: Cache coherence protocols are responsible for keeping data integrity in shared memory parallel architectures. They keep track of the state of all cache lines, identifying which cache lines are exclusive to a cache, shared between caches, and if the cache line has been written. Most coherence protocols employed in current architectures derive from the MOESI protocol. Coherence protocols are optimized by thread mapping because data that would be replicated in several caches in a bad mapping, thereby considered as *shared*, can be allocated in a cache shared by the threads that access the data, such that the data is considered as *private*. This is depicted in Figure 2.2, where 2 threads that share data are running in separate cores. When using a bad thread mapping, as in Figure 2.2a, the threads are running in cores that do not share any cache, and due to that the shared data is replicated in both caches, and the coherence protocol needs to send messages between the caches to keep data integrity. On the other

hand, when the threads use a good mapping, as in Figure 2.2b, the shared data is stored in the same cache, and the coherence protocol does not need to send any messages.

Reduction of cache misses: Since thread mapping influences the shared data stored in the cache memories, it also affects the cache misses. We identified three main types of cache misses that are reduced by an efficient thread mapping (CRUZ; DIENER; NAVAUX, 2012):

Capacity misses: In shared-memory applications, threads that access a lot of memory and share a cache evict cache lines from each other and cause capacity misses (ZHOU; CHEN; ZHENG, 2009). When mapping threads that share data to cores sharing a cache, we reduce competition for cache lines between cores that share a cache.

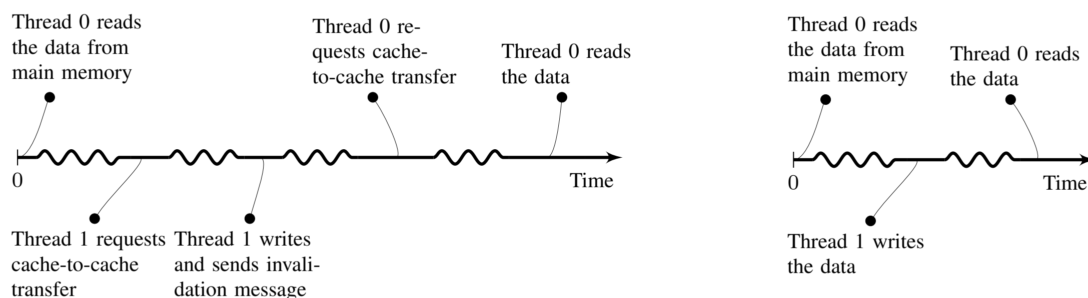
Invalidation misses: Cache coherence protocols send invalidation messages every time a write is performed in shared cache lines. In Figure 2.3a, we illustrate what happens



(a) Bad thread mapping. The cache coherence protocol needs to send messages to keep the integrity of data replicated in the caches, degrading the performance and energy efficiency.

(b) Good thread mapping. The data shared between the threads are stored in the same shared cache, such that the cache coherence protocol does not need to operate in them.

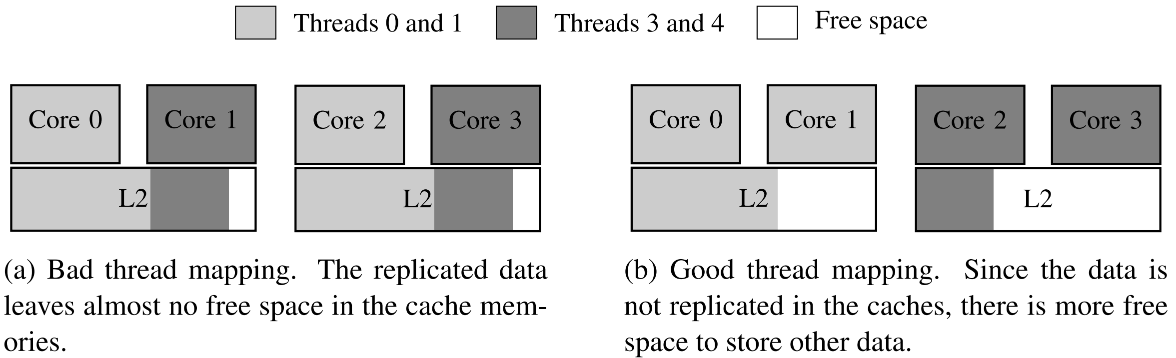
Figure 2.2: The relation between cache coherence protocols and sharing-aware thread mapping.



(a) Bad thread mapping. The cache coherence protocol needs to invalidate the data in the write operation, and then performs a cache-to-cache transfer for the read operation.

(b) Good thread mapping. The cache coherence protocol does not need to send any messages.

Figure 2.3: How a good mapping can reduce the amount of invalidation misses in the caches. In this example, core 0 reads the data, then core 1 writes, and finally core 0 reads the data again.



(a) Bad thread mapping. The replicated data leaves almost no free space in the cache memories.

(b) Good thread mapping. Since the data is not replicated in the caches, there is more free space to store other data.

Figure 2.4: Impact of sharing-aware mapping on replication misses.

when 2 threads that access one same cache line, and have a bad thread mapping (do not share a cache memory), perform the following actions: thread 0 reads the cache line, then thread 1 writes the cache line, and finally thread 0 reads the cache line again.

Since, the threads do not share a cache, when thread 1 writes, it misses the cache and thereby needs first to retrieve the cache line to its cache, and then invalidates the copy in the cache of thread 0. When thread 0 reads the data again, it will generate another cache miss, requiring a cache-to-cache transfer from the cache of thread 1.

On the other hand, when the threads share a common cache, as shown in Figure 2.3b, the write operation of thread 1 and the second read operation of thread 0 do not generate any cache miss. Summarizing, we reduce the number of invalidations of cache lines accesses by several threads, since these lines would be considered as private by cache coherence protocols (MARTIN; HILL; SORIN, 2012).

Replication misses: A cache line is said to be replicated when it is present in more than one cache. As stated in (CHISHTI; POWELL; VIJAYKUMAR, 2005), uncontrolled replication leads to a virtual reduction of the size of the caches, as some of their space would be used to store the same cache lines. By mapping applications that share large amounts of data to cores that share a cache, the space wasted with replicated cache lines is minimized, leading to a reduction of cache misses. In Figure 2.4, we illustrate how thread mapping affects cache line replication, in which two pairs of threads share data within each pair.

In case of a bad thread mapping, in Figure 2.4a, the threads that share data are mapped to cores that do not share a cache, such that the data needs to be replicated in the caches, leaving very little free cache space to store other data.

In case of a good thread mapping, in Figure 2.4b, the threads that share data are mapped to cores that share a common cache. In this way, the data is not replicated and there is much more free space to store other data in the cache.

Reduction of memory accesses to remote NUMA nodes: In NUMA systems, the time to access the main memory depends on the core that requested the memory access and the

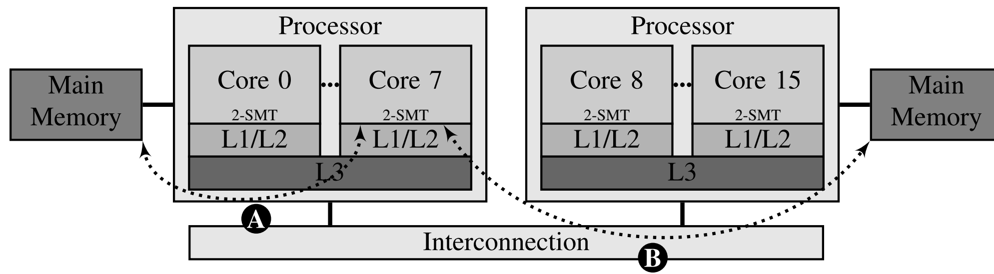


Figure 2.5: System architecture with two processors. Each processor consists of eight 2-way SMT cores and three cache levels.

NUMA node that contains the destination memory bank (RIBEIRO et al., 2009). We show in Figure 2.5 an example of NUMA architecture. If the core and destination memory bank belong to the same node, we have a *local* memory access, as in Figure 2.5 Ⓐ. On the other hand, if the core and the destination memory bank belong to different NUMA nodes, we have a *remote* memory access, as in Figure 2.5 Ⓑ. Local memory accesses are faster than remote memory accesses. By mapping the threads and data of the application in such a way that we increase the number of local memory accesses over remote memory accesses, the average latency of the main memory is reduced.

Better usage of interconnections: The objective is to make better use of the available interconnections in the processors. We can map the threads and data of the application in such a way that we reduce inter-chip traffic and use intra-chip interconnections instead, which have a higher bandwidth and lower latency. In order to reach this objective, the cache coherence related traffic, such as cache line invalidations and cache-to-cache transfers, has to be reduced. The reduction of remote memory accesses also improves the usage of interconnections.

2.1.1.2 Example of Shared Memory Architectures Affected by Memory Locality

Most architectures are affected by the locality of memory accesses. In this section, we present examples of such architectures, and how memory locality impact in their performance.

Intel Harpertown The Harpertown architecture (INTEL, 2008) is an Uniform Memory Access (UMA) architecture, where all cores have the same memory accesses latency to any main memory back. We illustrate the hierarchy of Harpertown in Figure 2.6. Each processor has 4 cores, where every core has private L1 instruction and data caches, and each L2 cache is shared by 2 cores. Threads running on cores that share a L2 cache (Figure 2.6 Ⓐ) have the highest data sharing performance, followed by threads that share data through the intrachip interconnection (Figure 2.6 Ⓑ). Threads running on cores that make use of the Front Side Bus (FSB) (Figure 2.6 Ⓒ) have the lowest data sharing performance.

To access the main memory, all cores also needs to access the FSB, which generates a bottleneck.

Intel Nehalem/Sandy Bridge Intel Nehalem (INTEL, 2010b) and Sandy Bridge (INTEL, 2012a), as well as their current successors, follow the same memory hierarchy. They are NUMA architectures, in which each processor in the system has one integrated memory controller. Therefore, each processor forms a NUMA node. Each core is 2-way SMT, using a technology called *Hyper-Threading*. The interchip interconnection is called *QuickPath Interconnect* (QPI). The memory hierarchy is the same as the ones illustrated in Figures 2.1 and 2.5.

AMD Abu Dhabi The AMD Abu Dhabi (AMD, 2012) is also a NUMA architecture. The memory hierarchy is depicted in Figure 2.7, using two AMD Opteron 6386 processors. Each processor forms 2 NUMA nodes, as each one has 2 memory controllers. There are 4 possibilities for data sharing between the cores. Every couple of cores share a L2 cache (Figure 2.7 Ⓐ). The L3 cache is shared by 8 cores (Figure 2.7 Ⓑ). Cores within a processor can also share data using the intrachip interconnection (Figure 2.7 Ⓒ). Cores from different processors need to use the interchip interconnection, called *HyperTrans-*

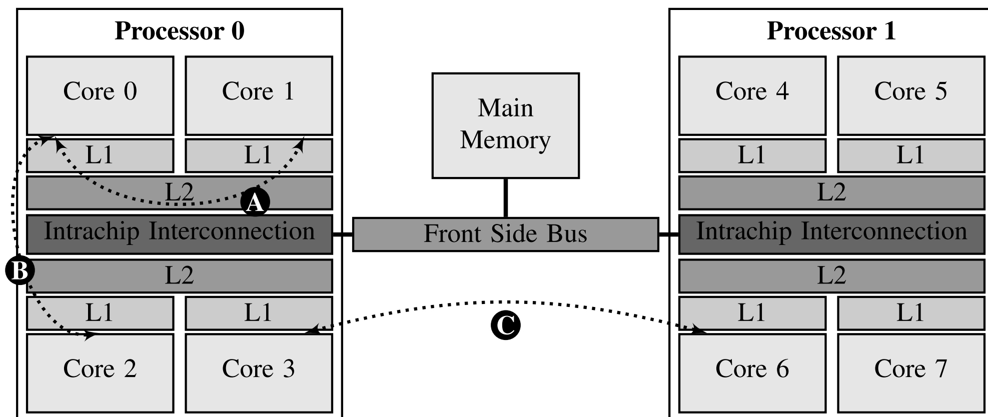


Figure 2.6: Harpertown architecture.

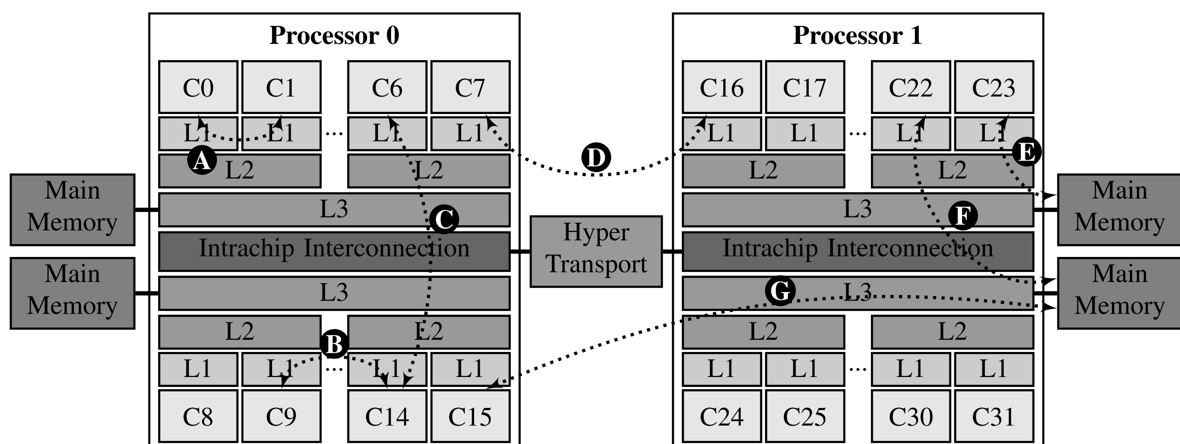


Figure 2.7: AMD Abu Dhabi architecture.

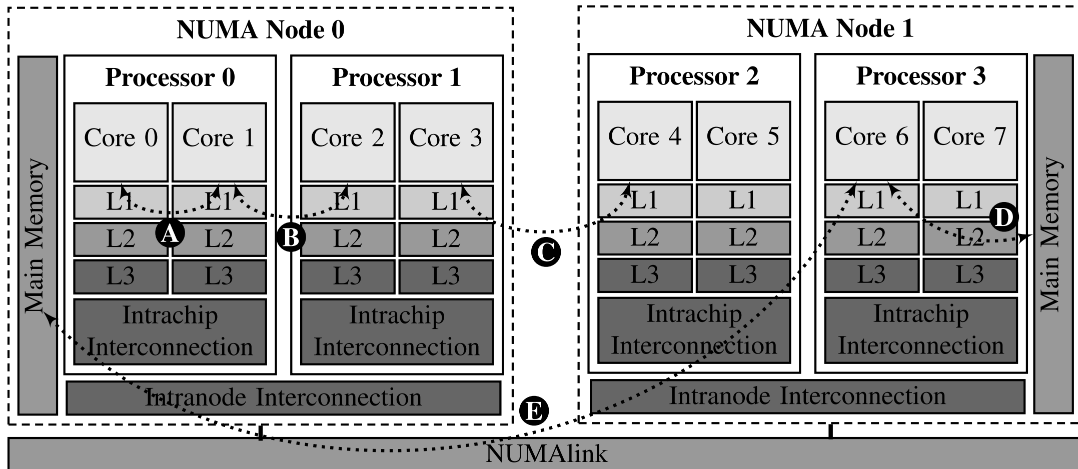


Figure 2.8: Intel Montecito/SGI NUMAlink architecture.

port (Figure 2.7 Ⓓ).

Regarding the NUMA nodes, cores can access the main memory with 3 different latencies. A core may access its local node (Figure 2.7 Ⓔ), using its local memory controller. It can access the remote NUMA node using the other memory controller within the same processor (Figure 2.7 Ⓕ). Finally, a core can access a main memory bank connected to a memory controller located in a different processor (Figure 2.7 Ⓖ).

Intel Montecito/SGI NUMAlink It is a NUMA architecture, but with characteristics different from the previously mentioned. We illustrate the memory hierarchy in Figure 2.8, considering that each processor is a dual-core Itanium 9030 (INTEL, 2010a). There are no shared cache memories, and each NUMA node contains 2 processors. Regarding data sharing, cores can exchange data using the intrachip interconnection (Figure 2.8 Ⓐ), the intranode interconnection (Figure 2.8 Ⓑ), or the SGI NUMAlink interconnection (Figure 2.8 Ⓒ). Regarding the NUMA nodes, cores can access their local memory bank (Figure 2.8 Ⓓ), or the remote memory banks (Figure 2.8 Ⓔ).

2.1.1.3 Locality in the Context of Network Clusters and Grids

Until this section, we analyzed how the different memory hierarchies influenced memory locality, and how improving memory locality benefits performance and energy efficiency. Although it is not the focus of this thesis, it is important to contextualize the concept of locality in network clusters and grids, which use message passing instead of shared memory. In such architectures, each machine represents a node, and the nodes are connected by routers, switches and network links with different latencies, bandwidths and topologies. Each machine is actually a shared memory architecture, as described in the previous sections, and can benefit from increasing memory locality.

Applications running in network clusters or grids are organized in processes instead of threads. 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. The discovery of the communication pattern of message passing based applications is straightforward compared to discovering the memory access pattern in shared memory based applications. This happens because the messages keep fields that explicitly identify the source and destination, which is an *explicit* communication, such that communication can be detected by monitoring the messages. On the other hand, in shared memory based applications, the focus of this thesis, the communication is *implicit* and happens when different threads access the same data. Implicit communication impose more challenges than explicit communication.

2.1.2 Parallel Applications and Sharing-Aware Thread Mapping

To analyze the memory locality in the context of sharing-aware thread mapping, we must investigate how threads of parallel applications share data, which we call *sharing pattern* (CRUZ et al., 2014a). For thread mapping, the most important information is the amount of memory accesses to shared data, and with which threads the data is shared. The memory address of the data are not relevant. In this section, we first expose some considerations about factors that influence thread mapping. Then, we analyze the data sharing patterns of parallel applications. Finally, we vary some parameters to verify how they impact the detected sharing patterns.

2.1.2.1 Considerations About the Sharing Pattern

We identified the following items that influence the detection of the data sharing pattern between the threads:

Dimension of the sharing pattern The sharing pattern can be analyzed by grouping different numbers of threads. To calculate the amount of data sharing between groups of threads of any size, the time and space complexity rises exponentially, which discourages the use of thread mapping for large groups of threads. Therefore, we evaluate the sharing pattern only between pairs of threads, generating a *sharing matrix*, which has a quadratic complexity.

Granularity of the sharing pattern (memory block size) To track which threads access each data, we need to divide the entire memory address space in blocks. A thread that access a memory block is called a *sharer* of the block. The size of the memory block directly influences the *spatial false sharing* problem (CRUZ; DIENER; NAVAUX, 2012), depicted in Figure 2.9.

Using a large block, as shown in Figure 2.9a, threads accessing the same memory block,

but different parts of it, will be considered sharers of the block. Therefore, the usage of large blocks increases the spatial false sharing.

Using blocks of lower size, as in Figure 2.9b, decreases the spatial false sharing, since it increases the possibility that access in different memory positions would be performed in different blocks.

In UMA architectures, the memory block size usually used is the cache line size, as the most important resource optimized by thread mapping is the cache. In NUMA architectures, the memory block is usually set to the size of the memory page, since the main memory banks are spread over the NUMA nodes using a page level granularity.

Data structure used to store the thread sharers of a memory block We store the threads that access each memory block in a vector we call *sharers vector*. There are 2 main ways to organize the sharers vector.

In the first way, we reserve one element for each thread of the parallel application, each one containing a time stamp of the last time the corresponding thread accessed the memory block. In this way, at a certain time T, a thread is considered as a sharer of a memory block if the difference between T and the time stored in the element of the corresponding thread is lower than a certain threshold. In other words, a thread is a sharer of a block it has accessed the block recently.

The second way to organize the structure is simpler, we store the IDs of the last threads that accessed the block. The most recent thread is stored in the first element of the vector, and the oldest thread in the last element of the vector. In this organization, a thread is considered a sharer of the memory block if its ID is stored in the vector.

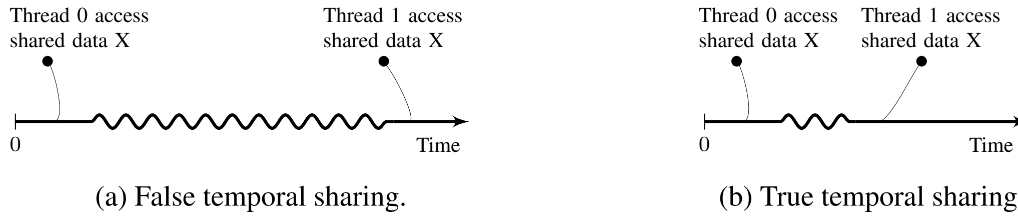
The first method to organize the thread sharers of a memory block has the advantage of storing information about all threads of the parallel application, as well as being able more flexible to adjust when a thread is considered a sharer. The second method has the advantage of being much simpler to be implemented. As the goal of this thesis is designing mechanisms that can be implemented in the hardware, we focus our analysis of the sharing pattern using the second method.

History of the sharing pattern Another important aspect of the sharing pattern is how much

(a) Low granularity. The data accessed by the threads belong to the same memory block.

(b) High granularity. The data accessed by the threads belong to different memory blocks.

Figure 2.9: How the granularity of the sharing pattern affects the spatial false sharing problem.



(a) False temporal sharing.

(b) True temporal sharing.

Figure 2.10: Temporal false sharing problem.

of the past events we should consider in each memory block. This aspect influences the *temporal false sharing* problem (CRUZ; DIENER; NAVAUX, 2012), illustrated in Figure 2.10. Temporal false sharing happens when two threads access the same memory block, but at very distant times during the execution, as shown in Figure 2.10a. To be actually considered as a data sharing, the time difference between the memory accesses to the same block should not be long, as in Figure 2.10b. Using the first method to organize the sharers of each memory block (explained in the previous item), the temporal sharing can be controlled by adjusting the threshold that defines if a memory access is recently enough to determine if a thread is sharer of a block. Using the second implementation, we can control the temporal sharing by setting how many thread IDs can be stored in the sharers vector. The more elements in the sharers vector, more thread IDs can be tracked, but increases the temporal false sharing.

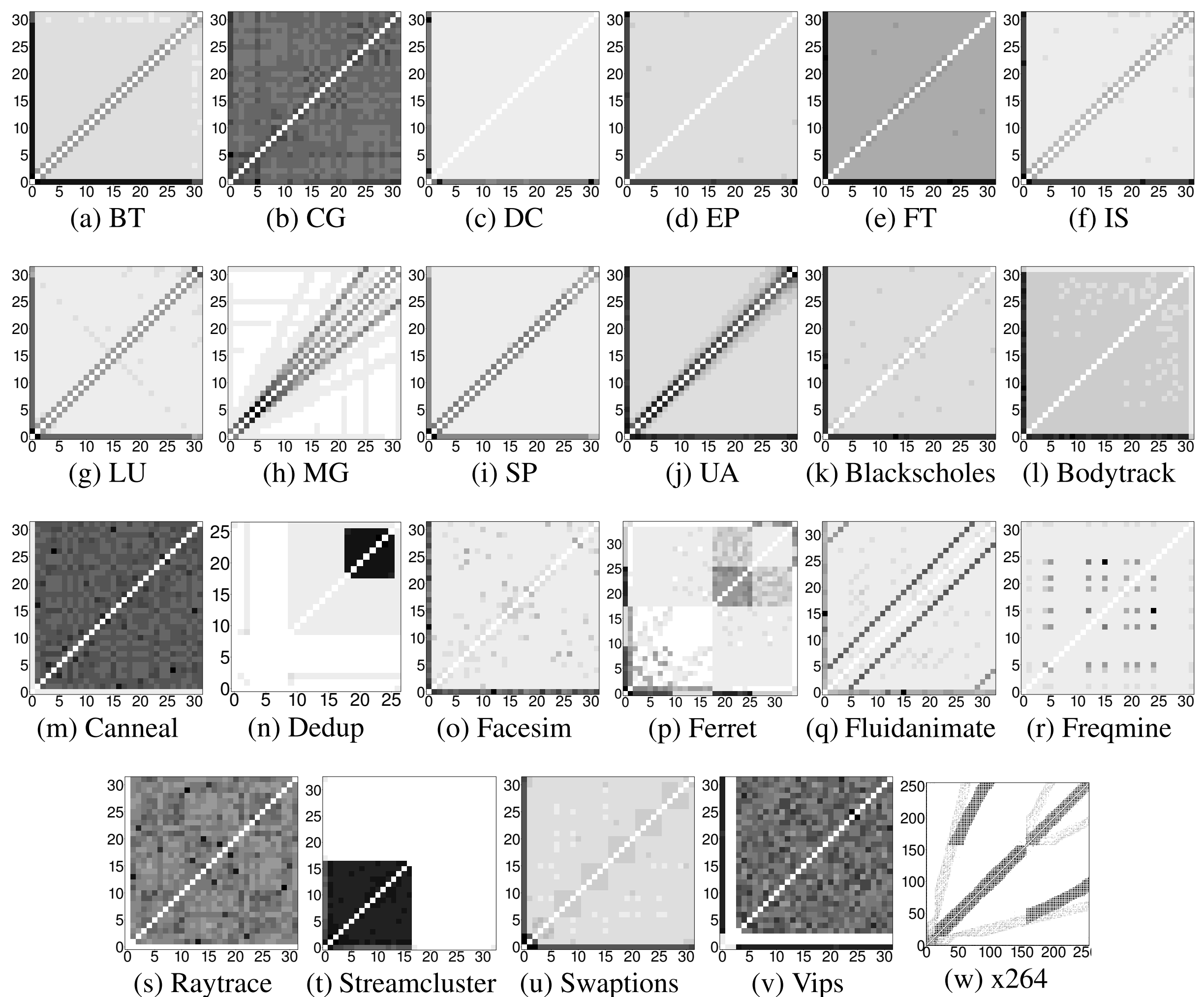
2.1.2.2 Sharing Pattern of Parallel Applications

The sharing matrices of some applications are illustrated in Figure 2.11, where axes represent thread IDs and cells show the amount of accesses to shared data for each pair of threads. Darker cells indicate more accesses. The sharing patterns were generated using a memory block size of 4 KBytes, and storing 2 sharers per memory block. Since the absolute values of the contents of the sharing matrices vary significantly between the applications, we normalized each sharing matrix to its own maximum value and color the cells according to the amount of data sharing. The data used to generate the sharing matrices was collected by instrumenting the applications using Pin (BACH et al., 2010), a dynamic binary instrumentation tool. The instrumentation code monitors all memory accesses, keeping track of which threads access each memory block. We analyze applications from the OpenMP NAS Parallel Benchmarks (JIN; FRUMKIN; YAN, 1999) and the PARSEC Benchmark Suite (BIENIA et al., 2008).

We were able to identify several sharing patterns in the applications. One pattern that is present in several applications is an *initialization* or *reduction* pattern, in which one of the threads, in this case thread 0, share data with all other threads, weather due to data initialization of the data or due to an aggregation of all data to generate a result. This patter is present in BT, DC, EP, FT, IS, LU, SP, UA, Blackscholes, Bodytrack, Facesim, Fluidanimate, Swaptions

and Vips. The reduction pattern cannot be optimized by a sharing-aware thread mapping, since there is no way to distribute the threads over the cores such that we increase memory locality. Another pattern that we identified consists of data sharing between *neighbor* threads, which is very common in applications that use a domain decomposition parallelization model, such as in IS, LU, MG, SP, UA and x264. In this pattern, data sharing can be optimized by mapping neighbor threads to cores close in the memory hierarchy. Some applications present both the initialization/reduction pattern and neighbor data sharing pattern, as BT, IS, LU, SP and UA.

Another pattern present in some applications is the sharing between *distant* threads. We can observe this pattern in MG, Fluidanimate and x264. Some applications present both the neighbor and distant sharing patterns, which represents a challenge for mapping algorithms. This happens because if you map neighbor threads close in the memory hierarchy, you lose the



(a) BT (b) CG (c) DC (d) EP (e) FT (f) IS
(g) LU (h) MG (i) SP (j) UA (k) Blackscholes (l) Bodytrack
(m) Canneal (n) Dedup (o) Facesim (p) Ferret (q) Fluidanimate (r) Freqmine
(s) Raytrace (t) Streamcluster (u) Swaptions (v) Vips (w) x264

Figure 2.11: Sharing patterns of parallel applications from the OpenMP NAS Parallel Benchmarks and the PARSEC Benchmark Suite. Axes represent thread IDs. Cells show the amount of accesses to shared data for each pair of threads. Darker cells indicate more accesses. The sharing patterns were generated using a memory block size of 4 KBytes, and storing 2 sharers per memory block.

opportunity to optimize data sharing between distant threads, and vice versa. Some applications present a *pipeline* data sharing model, as Dedup, Ferret and Streamcluster. We can identify this pattern by observing that threads that share data in the pipeline parallelization model form sharing clusters. In this pattern, data sharing can be optimized by mapping the threads that share data in the pipeline, which are the threads belonging to the same cluster, to cores close in the memory hierarchy. The pattern of Swaptions also resembles sharing clusters.

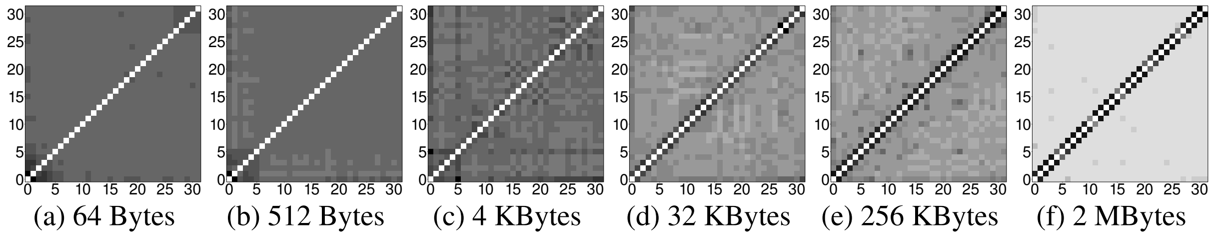
Finally, the last patterns that we observe are the *all-to-all* and *irregular* patterns. In the all-to-all pattern, all threads share a similar amount of data to all other threads, as in CG, DC, EP, FT, Blackscholes, Bodytrack, Canneal, Facesim and Raytrace. In the irregular pattern, we cannot clearly identify any of the previous characteristics, such as in Freqmine. In the all-to-all pattern, as well as the initialization or reduction patterns, sharing-aware thread mapping is not able to optimize data sharing, since there is no mapping that is able to improve locality. In all other patterns, neighbor, distant, pipeline and irregular patterns, sharing-aware thread mapping improves memory locality.

2.1.2.3 Varying the Granularity of the Sharing Pattern

To better understand how the memory block size affects the sharing-pattern, we varied the block size from 64 Bytes to 2 Mbytes (the previous analysis was performed using a 4 KBytes block size). We kept the number of thread sharers as 4 per memory block. Figures 2.12 to 2.17 show the sharing patterns of some applications using different block sizes. In CG (Figure 2.12), the sharing pattern shows a all-to-all pattern until a memory block size of 4 KBytes. With 32 KBytes blocks, the pattern begins to resemble a neighbor pattern, which is very evident using a 2 MBytes block size. Facesim (Figure 2.13) has a behavior similar to the one of CG. However, one interesting point of Facesim is that the number of data sharing neighbors in Facesim is higher with 256 KBytes blocks than with 2 MBytes blocks, forming sharing clusters similar to the ones of a pipeline parallelization model. Normally, we would expect the number of neighbors to increase with larger blocks, since it increases the probability that there will be more threads accessing each block.

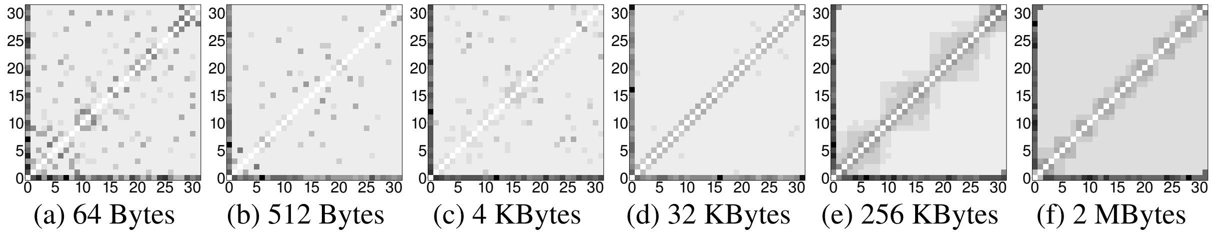
SP (Figure 2.14) is an application in which we can observe this expected behavior. Its sharing pattern is very stable until the usage of 32 KBytes blocks. However, with 256 KBytes blocks and, specially, 2 MBytes blocks, the number of neighbor threads sharing data increases significantly. Similarly, in LU (Figure 2.15), the number of neighbor threads sharing data is higher with larger block sizes. However, it is interesting to note that LU, with 64 Bytes and 512 Bytes blocks, also presents data sharing with distant threads. Fluidanimate (Figure 2.16) has a similar behavior: the sharing pattern when using larger block sizes present more data sharing between distant threads, which becomes very evident when using 32 KBytes blocks.

The last application in which we analyze the impact of the block size is Ferret (Figure 2.17). Until the usage of 256 KBytes blocks, the pipeline parallelization model is clearly seem due



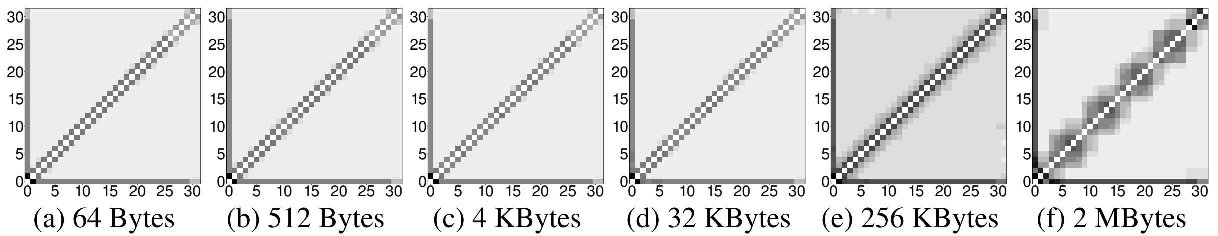
(a) 64 Bytes (b) 512 Bytes (c) 4 KBytes (d) 32 KBytes (e) 256 KBytes (f) 2 MBytes

Figure 2.12: Sharing patterns of CG varying the memory block size.



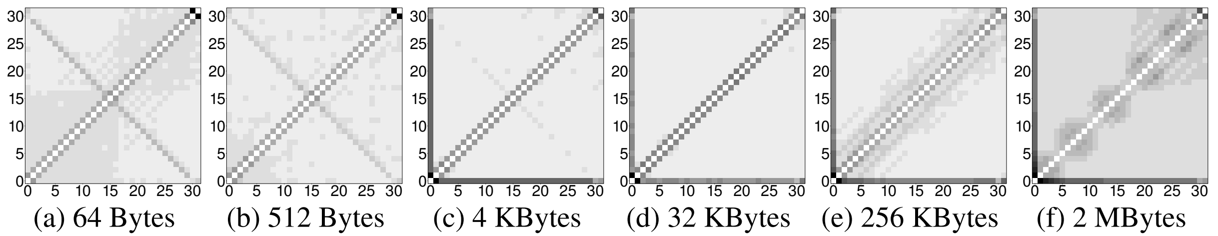
(a) 64 Bytes (b) 512 Bytes (c) 4 KBytes (d) 32 KBytes (e) 256 KBytes (f) 2 MBytes

Figure 2.13: Sharing patterns of Facesim varying the memory block size.



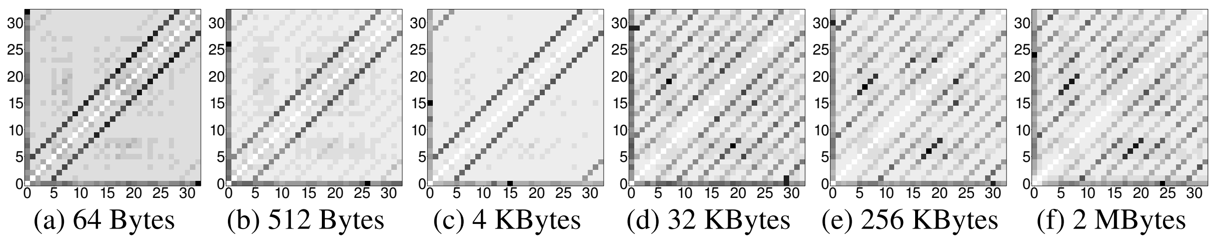
(a) 64 Bytes (b) 512 Bytes (c) 4 KBytes (d) 32 KBytes (e) 256 KBytes (f) 2 MBytes

Figure 2.14: Sharing patterns of SP varying the memory block size.



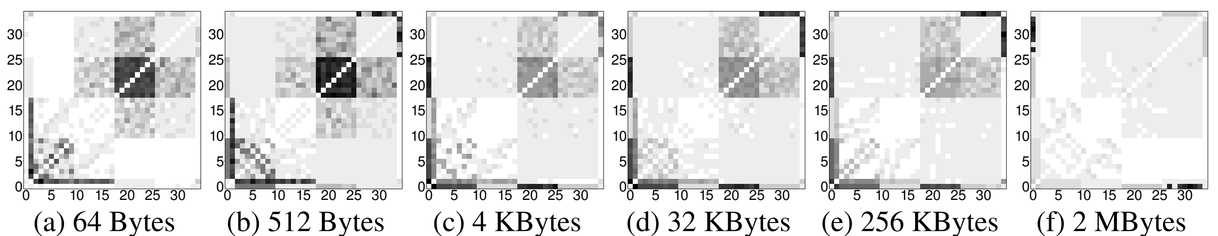
(a) 64 Bytes (b) 512 Bytes (c) 4 KBytes (d) 32 KBytes (e) 256 KBytes (f) 2 MBytes

Figure 2.15: Sharing patterns of LU varying the memory block size.



(a) 64 Bytes (b) 512 Bytes (c) 4 KBytes (d) 32 KBytes (e) 256 KBytes (f) 2 MBytes

Figure 2.16: Sharing patterns of Fluidanimate varying the memory block size.



(a) 64 Bytes (b) 512 Bytes (c) 4 KBytes (d) 32 KBytes (e) 256 KBytes (f) 2 MBytes

Figure 2.17: Sharing patterns of Ferret varying the memory block size.

to the sharing clusters. However, using 2 Mbytes blocks, the sharing pattern changes to an all-to-all pattern. This usually happens because the block size increased so much that most threads end up accessing most pages. One thing that is important to mention is that, although there are significant changes on the sharing pattern depending on the memory block size, that does not mean that some of these patterns are correct and some are wrong. It depends on the execution environment and resource the developer aims to optimize. For instance, if the goal is to optimize the cache usage, the sharing pattern using a cache size granularity is ideal. If the goal is to reduce remote memory access in a NUMA architecture, the detection using a page size granularity would be better.

2.1.2.4 Varying the Number of Sharers of the Sharing Pattern

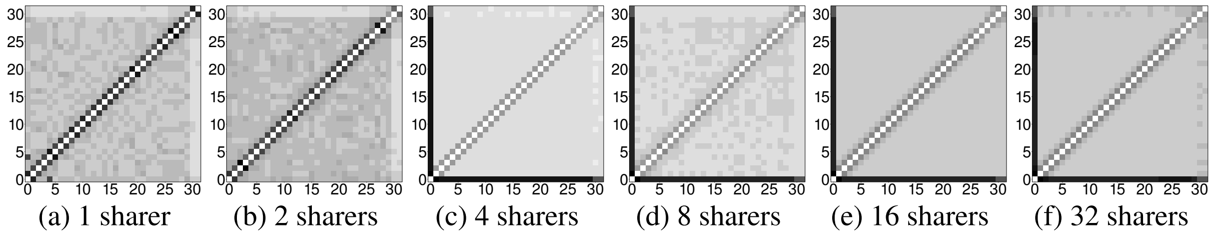
In the previous experiments, we analyzed the sharing pattern storing 4 sharer threads per memory block. To verify the impact of storing a different amount of sharers, we also performed experiments storing from 1 to 32 sharers per memory block. Figures 2.18 to 2.23 illustrate the sharing patterns of some applications varying the number of sharers. In BT (Figure 2.18), we can observe that, until the usage of 2 sharers, the initialization or reduction pattern is not present, but it shows up only when using 4 sharers. The reason is that most data sharing occurs between neighbor threads, such that the thread 0 gets removed from the sharers vector of a block when using less than 4 sharers. This same behavior happens in LU, Fluidanimate, Swaptions and Blackscholes (Figures 2.19, 2.20, 2.22 and 2.23).

In LU (Figure 2.19), until the usage of 4 sharers, we observe the data sharing between neighbors. With 8 sharers onwards, the data sharing between distant threads is also present. In Fluidanimate (Figure 2.20), we observe the opposite behavior, until 2 sharers there is a predominance of data sharing between distant threads, which decreases with more stored sharers. In Ferret (Figure 2.21), the sharing clusters are evident even with only 1 sharer, but with at least 4 sharers we can also observe sharing among different clusters. Swaptions (Figure 2.22) shows an all-to-all pattern until the usage of 2 sharers, and with more sharers it has sharering clusters.

The sharing pattern of Blackscholes (Figure 2.23) is an all-to-all pattern for all number of sharers evaluated. However, we can observe that with only one sharer, there is a lot of noise in the sharer pattern. This same characteristic was manifested in the other applications. Due to this reason, we discourage the usage of only 1 sharer per memory block. Developers must chose the amount of sharers depending on how much of past events they want to consider, as well as on how much overhead is tolerated, since the usage of more sharers increases the overhead.

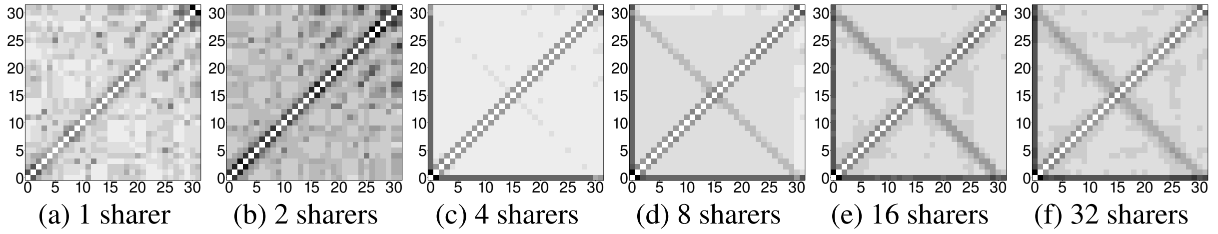
2.1.3 Parallel Applications and Sharing-Aware Data Mapping

For the data mapping, we need to know the amount of memory accesses from each thread (or NUMA node) to each page. In this way, each page can be mapped to the NUMA node



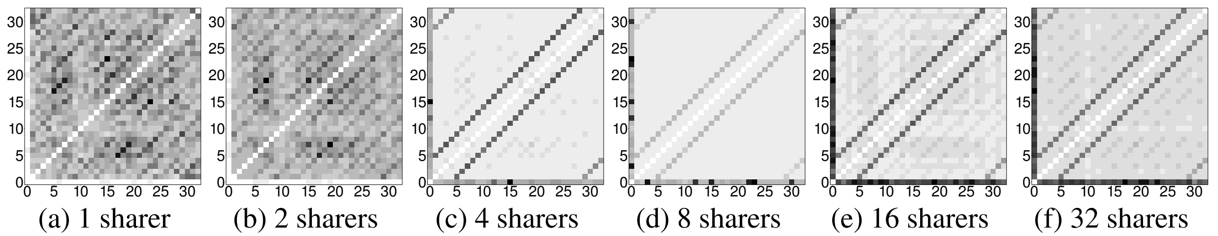
(a) 1 sharer (b) 2 sharers (c) 4 sharers (d) 8 sharers (e) 16 sharers (f) 32 sharers

Figure 2.18: Sharing patterns of BT varying the number of sharers.



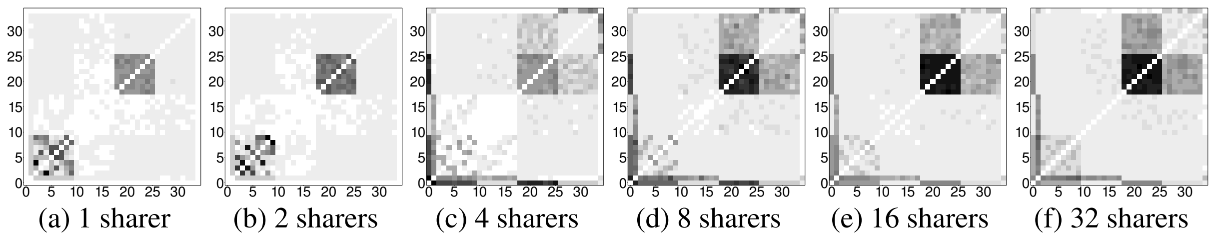
(a) 1 sharer (b) 2 sharers (c) 4 sharers (d) 8 sharers (e) 16 sharers (f) 32 sharers

Figure 2.19: Sharing patterns of LU varying the number of sharers.



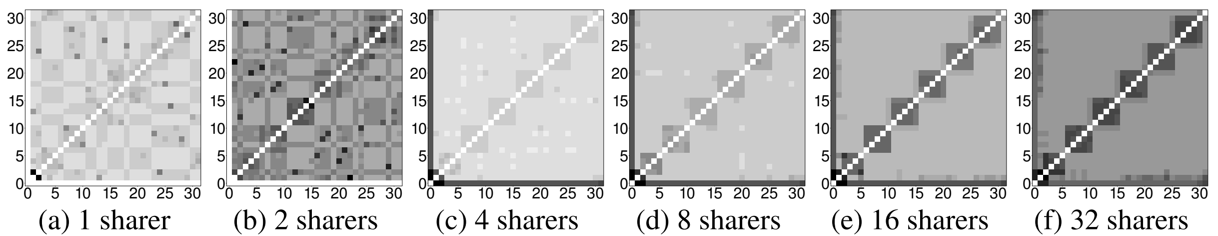
(a) 1 sharer (b) 2 sharers (c) 4 sharers (d) 8 sharers (e) 16 sharers (f) 32 sharers

Figure 2.20: Sharing patterns of Fluidanimate varying the number of sharers.



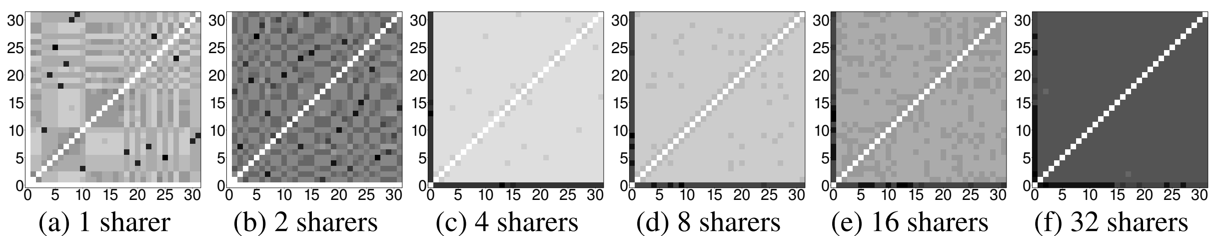
(a) 1 sharer (b) 2 sharers (c) 4 sharers (d) 8 sharers (e) 16 sharers (f) 32 sharers

Figure 2.21: Sharing patterns of Ferret varying the number of sharers.



(a) 1 sharer (b) 2 sharers (c) 4 sharers (d) 8 sharers (e) 16 sharers (f) 32 sharers

Figure 2.22: Sharing patterns of Swaptions varying the number of sharers.



(a) 1 sharer (b) 2 sharers (c) 4 sharers (d) 8 sharers (e) 16 sharers (f) 32 sharers

Figure 2.23: Sharing patterns of Blackscholes varying the number of sharers.

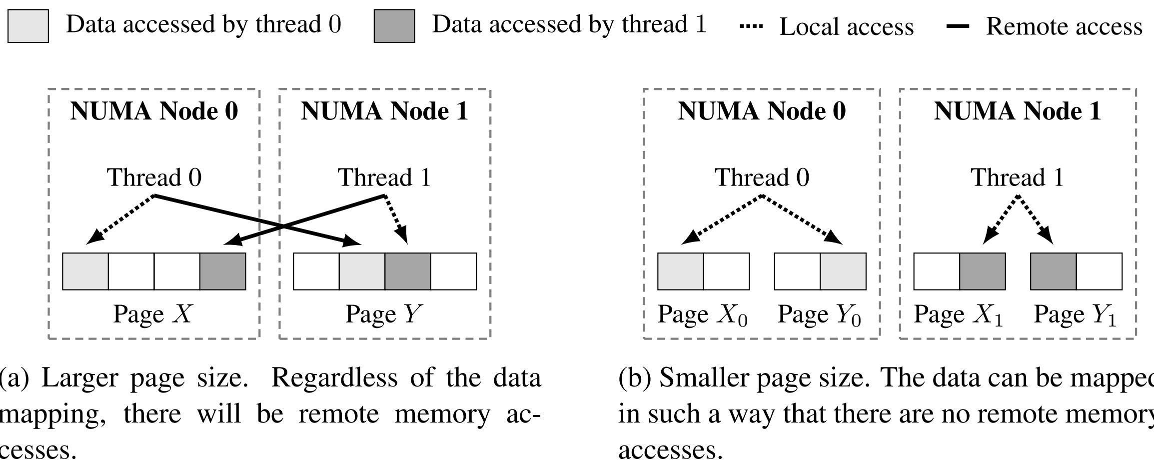
that performs most memory accesses to them. The goal is to reduce the amount of remote memory accesses, which are more expensive than local memory accesses. In this section, we first describe parameters that influence sharing-aware data mapping. Then, we present metrics to measure the impact of data mapping in an application, and use them to analyze parallel applications. Finally, we vary parameters to evaluate their impact in the data mapping.

2.1.3.1 Parameters that Influence Sharing-Aware Data Mapping

We identified the following parameters that influence sharing-aware data mapping:

Influence of the memory page size The memory page size is one of the parameters that most influence the potential for performance improvements from sharing-aware data mapping. This happens because, using higher page sizes, the probability of more threads sharing a page is increased. We can observe this behavior in Figure 2.24, where threads 0 and 1 access private data to each thread. Using a larger page size, as in Figure 2.24a, although the threads only access their private date, there is no way to map the pages in the NUMA nodes such that no remote memory accesses are performed. This happens because the private data of different threads are located in the same page. On the other hand, using a lower page size, as shown in Figure 2.24b, the data accessed by each thread are located in different pages. Thereby, the data can be mapped to NUMA node of the thread that access them, requiring no remote memory accesses.

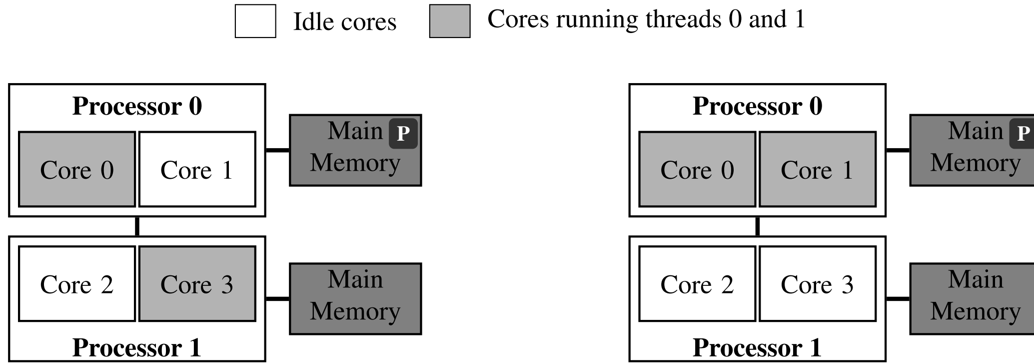
Influence of thread mapping Thread mapping influences sharing-aware data mapping in pages accessed by 2 or more threads. We illustrate this influence in Figure 2.25, where 2 threads access page P. When the threads are mapped to cores from different NUMA nodes, as in Figure 2.25a, one thread is able to access page P with local memory accesses, but the other thread has to perform remote memory accesses. However, when both threads



(a) Larger page size. Regardless of the data mapping, there will be remote memory accesses.

(b) Smaller page size. The data can be mapped in such a way that there are no remote memory accesses.

Figure 2.24: Influence of the page size on data mapping.



(a) Bad thread mapping. Thread 1 requires remote memory accesses to access page P.

(b) Good thread mapping. No remote memory access is necessary.

Figure 2.25: Influence of thread mapping on data mapping. In this example, threads 0 and 1 access page P.

are mapped to cores of the same NUMA node, as in Figure 2.25b, all memory accesses to page P are local, maximizing performance.

2.1.3.2 Analyzing the Data Mapping Potential of Parallel Applications

In sharing-aware data mapping, we need to analyze how the pages are individually shared. That is, for each page, we need to know which threads access it. To analyze the amount of accesses to private and shared data, we introduce a metric we call *exclusivity level* (DIENER et al., 2014). The exclusivity level of an application is higher when the amount of accesses to private data is higher than the amount of accesses to shared data. The exclusivity level of an application depends on the granularity in which we analyze the memory blocks. We perform our analysis in a page level granularity, using 4 Kbytes pages, which is the default page size in the x86-64 architecture.

The exclusivity level of a page is calculated using Equation 2.1, where $Access(p, t)$ is a function that returns the amount of memory accesses of a thread t to page p, $Access(p)$ is a function that returns a set with the amount of memory accesses from each thread to page p, N_T is the total number of threads, and max is a function that returns the maximum element of its input set. The lowest exclusivity a page can have is when it has the same amount of accesses from all threads. In this case, the page exclusivity would be $1/N_T$. The highest exclusivity of a page is when a page is only accessed by one thread, that is, a page is private. The exclusivity level of a private page is 1.

$$PageExclusivity(p) = 100 \cdot \frac{max(Access(p))}{\sum_{t=1}^{N_T} Access(p, t)} \tag{2.1}$$

In Figure 2.26, we show the amount of pages according to their exclusivity level in sev-

eral applications. Applications with the highest amounts of pages with high exclusivity are the ones with highest potential for performance improvements with sharing-aware data mapping. For instance, FT is the application with the highest amount of private pages (95.9% of private pages). Therefore, by mapping these pages to the NUMA node running the threads that accesses them, we are able to reduce the amount of remote memory accesses to *these pages* by up to 100%. Not only these private pages can be improved. For instance, in a page whose exclusivity level is 80%, the number of remote memory accesses to these pages can also be reduced by up to 80%. Only the pages that contain the lowest exclusivity level ($^1/_{N_T}$) are not affected by mapping. However, as we can observe in Figure 2.26, most pages have a high exclusivity level.

Although Figure 2.26 shows a detailed view of the pages, it can be difficult to have an overview of the entire application. To calculate the exclusivity level of the entire application, we can use Equation 2.2 or Equation 2.3. Equation 2.2 shows the average exclusivity level of all pages of the applications. Equation 2.3 also shows the average exclusivity level of all pages, but weighted to the amount of memory accesses to each page. Pages with more memory accesses have a higher influence in the application exclusivity level. Equation 2.3 should provide a better overview of the application, because the amount of memory accesses to each page can vary significantly.

$$AppExclusivityPages = \frac{\sum_{p=1}^{N_P} PageExclusivity(p)}{N_P} \tag{2.2}$$

$$AppExclusivityAccesses = \frac{\sum_{p=1}^{N_P} \sum_{t=1}^{N_T} Access(p,t) \cdot PageExclusivity(p)}{\sum_{p=1}^{N_P} \sum_{t=1}^{N_T} Access(p,t)} \tag{2.3}$$

The exclusivity level of a wide variety of applications is shown in Figure 2.27. We show the exclusivity level using both Equations 2.2 and 2.3. Applications with high exclusivity levels

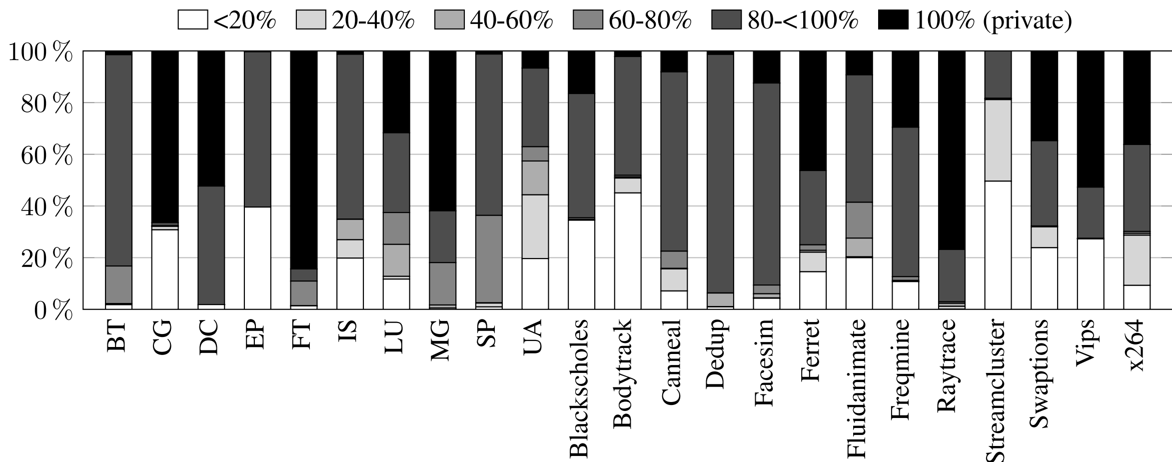


Figure 2.26: Page exclusivity of several applications in relation to the total number of memory accesses.

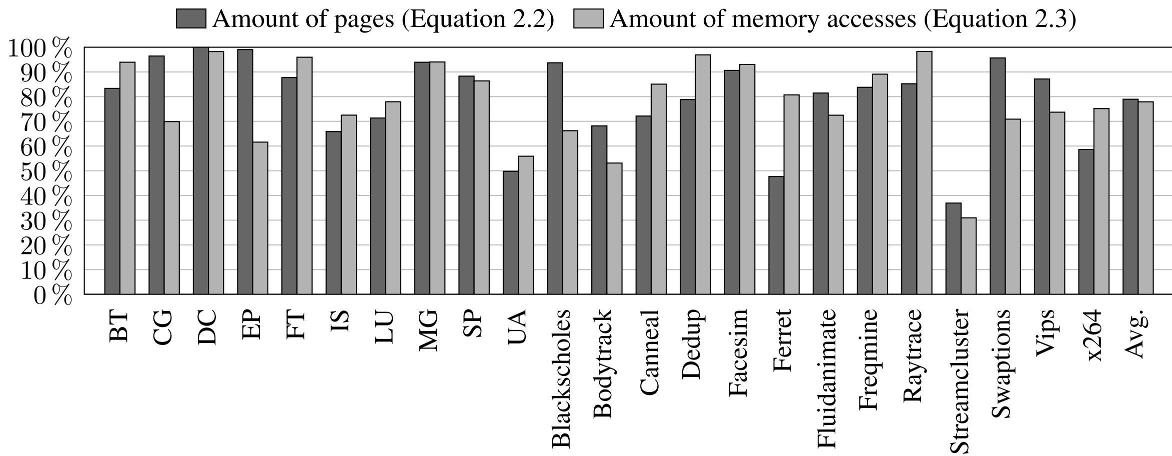


Figure 2.27: Exclusivity of several applications.

tend to benefit more from sharing-aware data mapping. This happens because, if a page with high exclusivity is mapped to the NUMA node of the threads that most access the page, the amount of remote memory accesses is reduced. On the other hand, if a page has a low exclusivity level, the amount of remote memory accesses would be similar regardless of the NUMA node used to store the page. In our experiment, the average exclusivity level of the applications, considering Equation 2.3, was 77.9%. This means that, on average, approximately 77.9% of the memory accesses can be improved by sharing-aware data mapping.

2.1.3.3 Influence of the Page Size on Sharing-Aware Data Mapping

As we previously explained, the page size influences sharing-aware data mapping. The larger the page size, the higher the probability of a page to be accessed by more threads. When this happens, the value of $max(p)$, in Equation 2.1, is lower in relation to the total amount of access to the corresponding page. In this way, the exclusivity level tends to decrease with larger pages. The previous experiments were based on a 4 KBytes page size. In Figure 2.28, we show the exclusivity level of several applications, using Equation 2.3, with page sizes up to 2 MBytes. We can clearly see the expected behavior: the exclusivity level decreases with larger page sizes. We can conclude with this experiment that the usage of lower page sizes are better for sharing-aware mapping.

2.1.3.4 Influence of Thread Mapping on Sharing-Aware Data Mapping

In the previous experiments, we analyzed the exclusivity level in relation to the amount of memory accesses from threads. In this way, the exclusivity level is independent from the thread mapping. However, if we consider a real world scenario, what influences the performance is the amount of memory accesses from each NUMA node, not from each threads. Therefore, we can change Equations 2.1 and 2.3 to Equations 2.4 and 2.5. In these modified equations,

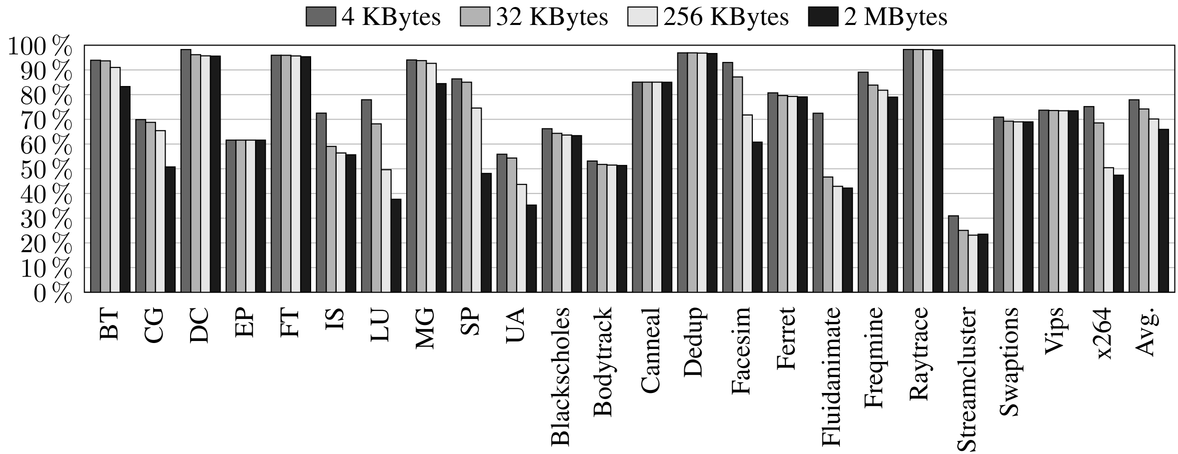


Figure 2.28: Page exclusivity of several applications in relation to the total number of memory accesses varying the page size.

$Access(p, n)$ is a function that returns the amount of memory accesses of NUMA node n to page p, $Access(p)$ is a function that returns a set with the amount of memory accesses from each node to page p, N_{nodes} is the total number of NUMA nodes, and max is a function that returns the maximum element of its input set. Using these equations, the lowest page exclusivity would be $1/N_{nodes}$, and the highest exclusivity of a page is 1.

$$PageExclusivityNode(p) = 100 \cdot \frac{max(Access(p))}{\sum_{n=1}^{N_{nodes}} Access(p, n)} \tag{2.4}$$

$$AppExclusivityNodeAccesses = \frac{\sum_{p=1}^{N_P} \sum_{n=1}^{N_{nodes}} Access(p, n) \cdot PageExclusivityNode(p)}{\sum_{p=1}^{N_P} \sum_{n=1}^{N_{nodes}} Access(p, n)} \tag{2.5}$$

In Figure 2.29, we show the exclusivity level of the applications considering the NUMA

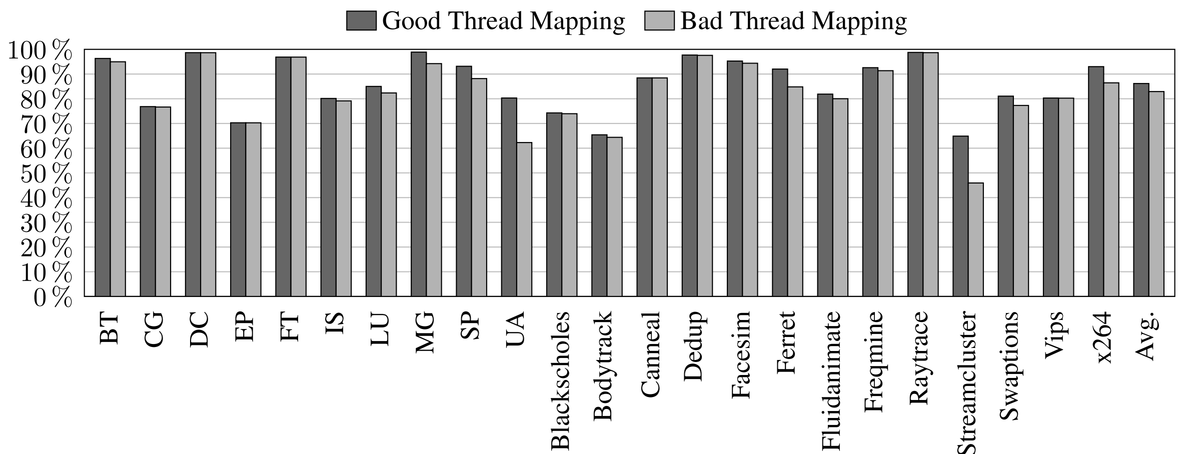


Figure 2.29: Page exclusivity of several applications in relation to the total number of memory accesses varying the thread mapping.

nodes (4 nodes), with 2 configurations. In the first configuration, threads that share large amounts of data are mapped to cores close to each other in the memory hierarchy. To generate this thread mapping, we apply a thread mapping algorithm (Appendix A) in the sharing matrices of the applications (Section 2.1.2.2). In the second configuration, we map the threads to cores distant in the memory hierarchy, by applying a modified version of the thread mapping algorithm used in the first configuration. We can observe, in Figure 2.29, the expected result: a better thread mapping results in a higher exclusivity level, and thereby in a higher potential for performance improvements from sharing-aware data mapping.

2.1.4 Evaluation with a Microbenchmark

To illustrate how sharing-aware thread mapping affects the memory locality, consider a producer-consumer situation in shared memory programs, in which one thread writes to an area of memory and another thread reads from the same area. The pseudo-code of this producer-consumer microbenchmark is illustrated in Algorithm 2.1. If the cache coherence protocol is based on invalidation, such as *MESI* or *MOESI* (STALLINGS, 2006), and the consumer and producer threads do not share a cache, an invalidation message is sent to the cache of the consumer every time the producer writes the data. As a result, after the invalidation, the consumer always receives a cache miss when reading, thereby requiring more traffic on the interconnections, since the cache of the consumer has to retrieve the data from the cache of the producer on every access. However, if the consumer and producer threads share a cache, the number of cache misses and interconnection traffic are reduced, since both the producer and the consumer access the data in the same cache, eliminating the need for invalidation messages and cache-to-cache transfers.

Algorithm 2.1: Producer Consumer.

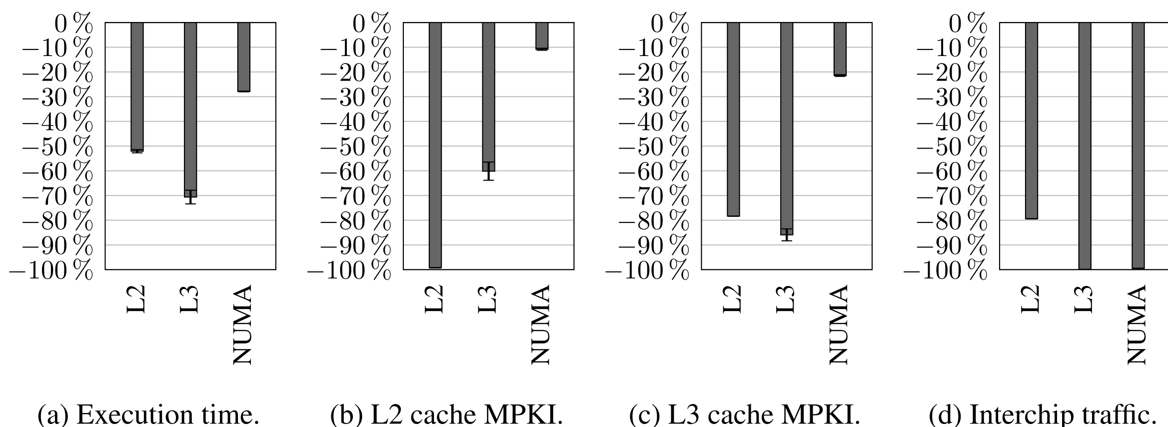
```
  begin Producer                              begin Consumer
      for s from 1 to NumberOfSteps do            for s from 1 to NumberOfSteps do
          for i from 1 to VectorSize do               Wait for producer signal;
              produce Vector[i];                      for i from 1 to VectorSize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

To understand how the memory locality is affected by sharing-aware data mapping, consider the same producer-consumer scenario, but that the amount of shared data is much larger, such that all memory accesses result in cache misses and must be resolved by the main memory. Also consider that the producer consumer threads are running on cores from the same NUMA node. If the shared data was allocated in a NUMA node different from the node where the threads are running, all memory accesses would be remote. On the other hand, if the shared data was

allocated in the same NUMA node where the threads are running, all memory accesses would be local.

We performed experiments with this producer-consumer microbenchmark in a machine with the memory hierarchy depicted in Figure 2.1. Details on the architecture are shown in Section 6.1.4. We measured execution time, L2 and L3 cache misses per thousand instructions (MPKI), and interchip interconnection traffic. The results are shown in Figure 2.30. We performed three types of experiments: *L2*, *L3* and *NUMA*. We execute each test 10 times, and show a 95% confidence interval calculated with the Student's t-distribution. In the *L2* and *L3* experiments, the size of the vector used by the producer-consumer threads was set to fit in the L2 and L3 caches, respectively. In the *NUMA* experiment, the vector size is much larger than the L3 cache. For each experiment, we executed with the best and worst possible mappings. The bar corresponds to the result of the best mapping normalized to the worst mapping. In the best mapping, the threads are mapped to virtual cores sharing the same L2 and L3 caches, and the vector to the NUMA node of the corresponding cores. In the worst mapping, no resources are shared.

The result of the *L2* experiment shows that, when the producer-consumer threads share a common L2 cache, the amount of L2 cache misses is reduced by almost 100%. This is the expected result, since, in the best mapping, the threads would always get a cache hit when reading and writing in the vector. In the *L3* experiment, L3 cache misses were reduced by 86.6%. The reduction was lower than in the *L2* experiment because there are several other cores sharing the same L3 cache, interfering in the results. The amount of interchip interconnection traffic was reduced by almost 100% in the *NUMA* experiment. For this particular application, the shared resource that provided the highest improvements when comparing the best and worst mappings was the L3 cache. Different applications may be more influenced by other resource. These results show that sharing-aware mapping is able to improve performance in parallel applications.



(a) Execution time. (b) L2 cache MPKI. (c) L3 cache MPKI. (d) Interchip traffic.

Figure 2.30: Producer-consumer microbenchmark results.

2.2 Related Work on Sharing-Aware Static Mapping

To statically map an application, the application must be executed inside controlled environments to gather information on the sharing pattern. 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, such as Qemu (BELLARD, 2005) and Bochs (SHWARTSMAN; MIHOCKA, 2008) only guarantee that the program output for a given input will be the same of the real machine. Simulators focus on implementing the virtual machine so that the behavior will be similar to the real machine. Simulators can be capable of running full systems and native operating systems, such as Simics (MAGNUSSON et al., 2002), MARSS (PATEL et al., 2011) and GEM5 (BINKERT et al., 2011), or can be capable of simulating only user level binaries, such as Multi2sim (SWAMY; UBAL, 2014). 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 tools presented in the previous paragraphs are controlled environments that can be used to monitor the behavior of parallel applications. They demand a lot of computational resources, drastically increasing the overhead of mapping approaches. In the rest of this section, we present some related work about sharing-aware mapping that use these controlled environments to gather information to map the threads and data of parallel applications. We describe mechanisms that perform thread and data mapping separately, and then mechanisms that perform both mappings together.

2.2.1 Static Thread Mapping

In Barrow-Williams, Fensch and Moore (2009), a technique to collect the sharing 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. In the second step, the memory traces are analyzed to determine the sharing pattern of the applications. When analyzing the memory trace, they do not consider consecutive accesses to the same address to ignore memory accesses that occurred due to register number constraints. The authors categorize the memory accesses considering the amount of read and write operations performed. In Bienia, Kumar and Li (2008), the authors use Pin to analyze parallel applications, evaluating the amount of memory shared by the threads and the size of the working set (TANENBAUM, 2007). As the main goal of the work was just to characterize the sharing pattern, the authors did not ran any performance tests using the collected data.

The works described in Diener et al. (2010) and Cruz, Alves and Navaux (2010) use Simics to discover the sharing pattern of parallel applications. The authors used the collected informa-

tion to map the threads to cores according to the amount of sharing. These methods can provide a high accuracy, as they have access to information of the entire execution of the application. However, they impose a high overhead, since they require the generation and analysis of memory traces. By depending on memory traces, they are limited by applications whose behavior do not change between or during the execution, which would invalidate the previous generated memory trace. Also, these thread mapping mechanisms are not able to reduce the number of remote memory accesses in NUMA architectures.

2.2.2 Static Data Mapping

The work Ribeiro et al. (2009) describe a user level framework named Memory Affinity Interface (MAi), which allows data mapping control on Linux based NUMA platforms. MAi simplifies data mapping management issues, since it offers a wide set of memory allocation policies to control data distribution. MAi can only be used in applications written in the C or C++ languages. Parallel applications written on other programming languages are not supported. Minas (RIBEIRO et al., 2011) is a framework for data mapping that uses MAi to control data mapping, and implements a preprocessor that analyzes the source code of an application to determine a suitable data mapping for a target architecture. After the analysis, instrumentation code to perform data mapping is automatically inserted in the application.

A hardware based profile mechanism is described in Marathe, Thakkar and Mueller (2010). The authors use features from the performance monitoring unit of the Itanium-2 processor that provide samples of memory addresses accessed by the running application. The samples are from long latency load operations and misses on the TLB. In the software level, they capture the memory addresses samples provided by the hardware and associate it with the thread that performed the memory access, generating a memory trace. Afterwards, the memory trace is analyzed in order to select the best mapping of memory pages to the NUMA nodes of the architecture. In future executions of the same application, its pages are mapped according to the detected mapping. It is not clear in the paper how the authors handle the mapping of dynamic allocated memory.

These mechanisms that perform only data mapping are not capable of reducing the number of cache misses, since they do not consider the memory sharing between the threads. They also fail to reduce the number of remote memory accesses to shared pages. If a page is accessed by only one thread, it should be mapped to the NUMA node of the thread that is accessing it. On the other hand, consider a scenario in which a given page is accessed by more than one thread, thereby being shared. If the threads are being executed on different NUMA nodes, a data mapping only mechanism is not able to reduce the amount of remote memory accesses. However, if a mechanism takes into account the sharing pattern and maps most of the threads that share pages in the same NUMA nodes, the amount of remote accesses is reduced.

2.2.3 Combined Static Thread and Data Mapping

In Cruz et al. (2011), the Minas data mapping framework (RIBEIRO et al., 2011) was extended with a sharing-aware thread mapping mechanism. Information on the sharing pattern was gathered with the help of the Simics simulator to generate memory traces. Two metrics were evaluated when generating the sharing patterns: amount of shared memory and amount of accesses to shared memory, which resulted in different patterns for most of the evaluated applications. As previously stated, the requirement of memory traces represents a high overhead. Dynamic allocated memory also imposes a challenge to static data mapping, since the memory addresses and the threads that perform memory allocation may change on future executions, which would make the previously detected information useless.

2.3 Related Work on Sharing-Aware Online Mapping

In online mapping mechanisms, the runtime environment, whether it is located in the operating system or in a user level library, must be capable of detecting sharing information and perform mapping during the execution of the application. The information can be provided by the hardware, for instance from hardware performance counters, or by software instrumentation code. Several libraries, such as Perfmon[1] and Papi (JOHNSON et al., 2012), provide easy access to to hardware performance counters that can be used by mapping mechanisms. As in the previous section, we describe mechanisms that perform thread and data mapping separately, and then mechanisms that perform both mappings together.

2.3.1 Online Thread Mapping

The work described in Azimi et al. (2009) schedule threads by making use of hardware counters present in the Power5 processor. The hardware counter used by them stores the memory address of memory accesses resolved by remote cache memories. Other memory accesses are not taken into account. Traps to the operating system are required to capture each sample of memory address. To reduce the overhead generated by the traps, they only enable the mapping mechanism for an application when the core stall time due to memory accesses exceeds a certain threshold, which decreases the accuracy due to the lower number of samples. A list of accessed memory addresses is kept for each thread of the parallel application, and their contents are compared to generate the sharing pattern, which is also an expensive procedure.

Another mechanism that samples memory addresses is described in Diener, Cruz and Navaux (2013), called SPCD, where page faults are artificially added during the execution, allowing the operating system to capture memory addresses accessed by each thread. Page faults were introduced by clearing the page present bit of several pages in the page table. Besides the

[1] http://http://perfmon2.sourceforge.net/

traps to the operating system due to the added page faults, TLB shootdowns are required every time a page present bit is cleared, which can harm the performance (VILLAVIEJA et al., 2011). Only a small number of page faults are added to avoid a high overhead. CDSM (DIENER et al., 2015a) is an extension of SPCD to support applications with multiple processes. These mechanisms based on sampling do not have a high accuracy, since most memory operations are not considered, which results in incomplete sharing patterns.

In Cruz, Diener and Navaux (2012), Cruz, Diener and Navaux (2015), the sharing pattern is detected by comparing the entries of all TLBs. The mechanism can be implemented in current software-managed TLBs architectures, but requires the addition of one instruction to access the TLB entries in hardware-managed TLBs. The comparison of TLBs can demand a lot of time if there are too many cores in the system. To reduce the overhead, the comparison of TLB entries can be done less frequently, which reduces the accuracy. Hardware changes are also necessary in Cruz et al. (2014a), in which a sharing matrix is kept by the hardware by monitoring the source and destination of invalidation messages of cache coherence protocols. One disadvantage of both mechanisms is that their sharing history is limited to the size of the TLBs and cache memories, respectively. Therefore, their accuracy is reduced when an application uses too much memory.

The usage of the instructions per cycle (IPC) metric to guide thread mapping is evaluated in Autopin (KLUG et al., 2008). Autopin itself does not detect the sharing pattern, it only verifies the IPC of several mappings fed to it and executes the application with the thread mapping that presented the highest IPC. In Radojković et al. (2013), the authors propose BlackBox, a scheduler that, similarly to Autopin, selects the best mapping by measuring the performance that each mapping obtained. When the number of threads is low, all possible thread mappings are evaluated. When the number of threads makes it unfeasible to evaluate all possibilities, the authors execute the application with 1000 random mappings to select the best one. These mechanisms that rely on statistics from hardware counters take too much time to converge to an optimal mapping, since they need to first check the statistics of the mappings. The convergence usually is not possible because amount of possible mappings is exponential in the number of threads. Also, these statistics do not accurately represent sharing patterns. As mentioned in Section 2.2.1, mechanisms that only perform thread mapping are not capable of decreasing the number of remote memory accesses in NUMA architectures.

2.3.2 Online Data Mapping

Traditional data mapping strategies, such as *first-touch* and *next-touch* (LöF; HOLMGREN, 2005), have been used by operating systems to allocate memory on NUMA machines. In the case of first-touch, a memory page is mapped to the node of the thread that causes its first page fault. This strategy affects the performance of other threads, as pages are not migrated during execution. When using next-touch, pages are migrated between nodes according to memory

accesses to them. However, if the same page is accessed from different nodes, next-touch can lead to excessive data migrations. Problems also arise in the case of thread migrations between nodes (RIBEIRO et al., 2009). The NUMA Balancing policy (CORBET, 2012b) was included in more recent versions of the Linux kernel. In this policy, the kernel introduces page faults during the execution of the application to perform lazy page migrations, reducing the amount of remote memory accesses. However, NUMA Balancing does not detect the sharing pattern between the threads. A previous proposal with similar goals was AutoNUMA (CORBET, 2012a).

A page migration mechanism that uses queuing delays and row-buffer hit rates from memory controllers is proposed in Awasthi et al. (2010). 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 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 online page migration during the execution of the application. 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 hit rate and queuing delays of the memory controller. The major problem of this work is that the information about which data each thread is not considered. The page to be migrated is randomly chosen, and may lead to an increase on the number of remote accesses.

The work described in Marathe and Mueller (2006) make use of the PMU of Itanium-2 to generate samples of memory addresses accessed by each thread. Their profiling mechanism imposes a high overhead, as it requires traps to the operating system on every high latency memory load operation or TLB miss. Hence, they enable the profiling mechanism just during the beginning of each application, losing the opportunity to handle changes during the execution of the application. They also suggest that the programmer of the application should instrument the source code of the application to provide to the operating system the areas of the application that should have their memory accesses monitored. Similarly, Tikir and Hollingsworth (2008) use UltraSPARC III hardware monitors to guide data mapping. However, their proposal is limited to architectures with software-managed TLBs. As explained in Section 2.2.2, mechanisms that do not perform thread mapping are not able to reduce cache misses and remote memory accesses to pages accessed by several threads.

The Carrefour mechanism (DASHTI et al., 2013) uses Instruction-Based Sampling (IBS), available on AMD architectures, to detect the memory access behavior and keeps a history of memory accesses to limit unnecessary migrations. Additionally, it allows replication of pages that are mostly read in different NUMA nodes. Due to the runtime overhead, Carrefour has to limit the number of samples they collect and the number of pages they characterize, tracking at most 30,000 pages.

2.3.3 Combined Online Thread and Data Mapping

A library called ForestGOMP is introduced in Broquedis et al. (2010a). This library integrates into the OpenMP runtime environment and gathers information about the different parallel sections of the applications. The threads of parallel applications are scheduled such that threads created by the same parallel region, which usually operate in the same dataset, execute on cores nearby in the memory hierarchy, decreasing the number of cache misses. To map data, the library depends on hints provided by the programmer. The authors also claim to use statistics from hardware counters, although they do not explain this in the paper. ForestGOMP only works for OpenMP based applications. Also, the need for hints to perform the mapping may be a burden to the programmer.

The kernel Memory Affinity Framework (kMAF) (DIENER et al., 2014) uses page faults to detect the memory access behavior. Whenever a page fault happens, kMAF verifies the ID of the thread that generated the fault, as well as the memory address of the fault. To increase its accuracy, kMAF introduces additional page faults. These additional page faults impose an overhead to the system, since each fault causes an interrupt to the operating system. Like Carrefour, kMAF has a limited access to the number of samples of memory accesses to control the overhead, harming the accuracy of the detected memory access behavior.

2.4 Discussion on Sharing-Aware Mapping and Related Work

In this chapter, we analyzed the relation between sharing-aware mapping, computer architectures and parallel applications. We observed that mapping is able to improve performance and energy efficiency in current architectures due to an improved memory locality, reducing cache misses and interchip interconnection usage. In the context of parallel applications, thread mapping affects their performance only in applications whose threads present different amount of data sharing among themselves, such that performance can be improved by mapping threads that share data in cores nearby in the memory hierarchy. Regarding data mapping, parallel applications whose memory pages presented a high amount of accesses from a single thread (or a single NUMA node) should benefit from an improved mapping of pages to NUMA nodes. The related work proved to have a wide variety of solutions, with very different characteristics, in which we summarize in Table 2.1. Analyzing the impact of mapping in the architectures and applications, as well the related work, we identified the following characteristics that would be desirable for a mechanism that performs sharing-aware mapping:

Perform both thread and data mapping As we observed, thread and data mapping influences the performance in different ways, such that they can be combined for maximum performance. Most of the related work perform only thread or data mapping alone.

Support dynamic environments Several related work required a previous analysis of the par-

	Thread mapping	Data mapping	Support dynamic environments	Dynamic memory allocation	Has access to memory addresses	Low overhead	Low trade-off accuracy×overhead	No manual source code modification	Do not require sampling
Barrow-Williams, Fensch and Moore (2009)	✓				✓			✓	✓
Bienia, Kumar and Li (2008)	✓				✓			✓	✓
Diener et al. (2010)	✓				✓			✓	✓
Cruz, Alves and Navaux (2010)	✓				✓			✓	✓
Ribeiro et al. (2009)		✓		✓	✓	✓			
Ribeiro et al. (2011)		✓			✓	✓		✓	
Marathe, Thakkar and Mueller (2010)		✓			✓			✓	✓
Cruz et al. (2011)	✓	✓			✓	✓		✓	
Azimi et al. (2009)	✓		✓		✓	✓		✓	
SPCD (DIENER; CRUZ; NAVAUX, 2013)	✓		✓		✓	✓		✓	
Cruz, Diener and Navaux (2015)	✓		✓		✓	✓		✓	
Cruz et al. (2014a)	✓		✓		✓	✓	✓	✓	✓
Autopin (KLUG et al., 2008)	✓		✓			✓		✓	
BlackBox (RADOJKOVIć et al., 2013)	✓		✓			✓		✓	
NUMA Balancing (CORBET, 2012b)		✓	✓	✓	✓	✓		✓	
Awasthi et al. (2010)		✓	✓	✓		✓	✓	✓	
Marathe and Mueller (2006)		✓			✓				
Carrefour (DASHTI et al., 2013)		✓	✓	✓	✓	✓		✓	
ForestGOMP (BROQUEDIS et al., 2010a)	✓	✓	✓	✓		✓			
kMAF (DIENER et al., 2014)	✓	✓	✓	✓	✓	✓		✓	

Table 2.1: Summary of related work.

allel application, not supporting applications that change the behavior between different executions. Also, such mechanisms may present issues to support the wide variety of parallel architectures, since an analysis considering a architecture may not be usable for another architecture.

Support dynamic memory allocation Thread mapping mechanisms do not apply to this characteristic. Regarding data mapping, several mechanisms perform the analysis using memory traces or with the compiler, which may not be enough to detect information for memory that is dynamically allocated. Also, dynamic allocated memory can change addresses between executions, which can invalidate previously generated information.

Has access to memory addresses In order to achieve a high accuracy, mechanisms should have access to the memory addresses being accessed by the threads. Mechanisms that analyze the behavior solely using statistics such as cache misses or instructions per cycle have a low accuracy.

Low overhead Several mechanisms require memory traces, which are expensive to generate and analyze.

Low trade-off between accuracy and overhead There are mechanisms that, when increasing the accuracy, the overhead drastically increases. Such mechanisms usually are based on sampling of memory addresses, such that the overhead is proportional to the amount of samples used.

No manual source code modification Mechanisms should not depend on source code modification, which could impose a burden on programmers. The wide variety of architectures aggravates the burden.

Do not require sampling Mechanisms that sample memory addresses can have a lower accuracy since the samples used may not characterize the behavior of the application correctly. Also, such mechanisms usually depend on sampling because their tracking method impose a high overhead. The usage of sampling can reduce the response time of the mechanism.

As a general comparison, we can observe that most related work either perform thread or data mapping, but not both of them together. In the case of thread mapping mechanisms, they are not able to reduce the amount of remote memory accesses in NUMA architectures. On the other hand, data mapping mechanisms are not able to reduce cache misses or correctly handle the mapping of shared pages. The mechanisms we described that perform both mappings together have several disadvantages. Cruz et al. (2011) is a static mechanism, depending on information from previous executions thereby being limited to applications whose behavior do not change between or during the execution. ForestGOMP (BROQUEDIS et al., 2010a) is online, but require hints from the programmer to work properly. kMAF (DIENER et al., 2014) use sampling and has a high overhead when we increase the amount of samples to achieve a higher accuracy. Other mechanisms rely on indirectly statistics obtained by hardware counters, which do not accurately represent the memory access behavior of parallel applications. Several proposals require specific architectures, APIs or programming languages to work, limiting their applicability.

With the analysis of the related work, we can conclude that currently there is no online mechanism that can be applied to any shared memory based application, in which increasing its accuracy does not drastically increases its overhead. Our proposals fulfill this gap of the related work. We perform both thread and data mapping to achieve improvements in terms of cache misses, remote memory accesses and interconnection usage. We implement our proposals directly in the MMU of the architecture, which allows us to keep track of many more memory accesses than the related work in a non intrusive way. In this way, we achieve a much higher accuracy in the detected patterns, while keeping a low overhead. The next chapters describe our proposals in detail.

3 LAPT – LOCALITY-AWARE PAGE TABLE

Our first proposal is a mechanism called *Locality-Aware Page Table* (LAPT) (CRUZ et al., 2014b). As explained in Section 1.3, the MMU of current processors that support virtual memory translate the virtual memory addresses to physical memory addresses for all memory accesses. In order to enable operating systems to perform thread and data mapping, LAPT uses the virtual memory implementation of current architectures. LAPT implements changes to the virtual memory subsystem on both the hardware and software levels. On the hardware level, LAPT keeps track of all TLB misses, as shown in Figure 3.1, detecting the data sharing between threads and registering a list of the latest threads that accessed each page in fields introduced in the corresponding page table entry. On the software level, the operating system maps threads to cores based on the detected sharing pattern, and maps pages to NUMA nodes by checking which threads accessed each page. A detailed representation of how LAPT works can be found in Figure 3.2.

We detail LAPT in this chapter. We first describe the data structures introduced by LAPT. Then, we explain how the data sharing is detected in hardware. Afterwards, we describe how the sharing information is employed to map threads and data. Lastly, we discuss the overhead of LAPT.

3.1 Structures Introduced by LAPT

LAPT adds new control registers to the architecture, and stores control data in the main memory. The following structures were added to the main memory:

Page Table In the page table, for each entry of the last level page table, we add a Sharers Vector *(SV)* and a NUMA Vector *(NV)*. The Sharers Vector holds the identifier of the last threads that accessed each page, and is kept by the hardware. The NUMA Vector holds

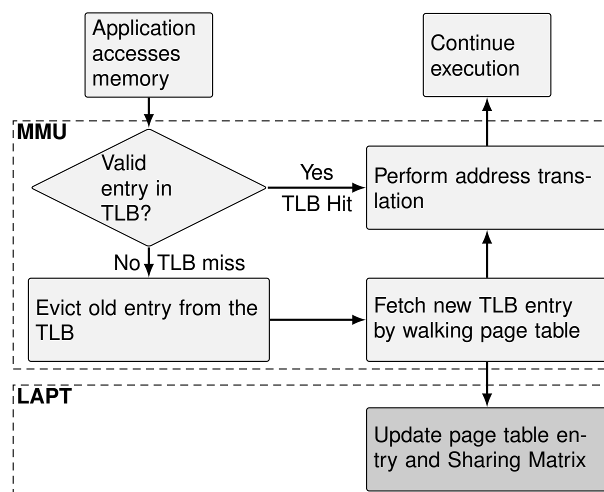


Figure 3.1: Overview of the MMU and LAPT.

the affinity of each NUMA node to the page, and is kept by the operating system.

Sharing Matrix Stores the affinity between each pair of threads. The size of of the Sharing Matrix is $nt \times nt$, where nt is the number of threads per parallel application supported by LAPT. This structure is updated by the hardware and used by the operating system to calculate the thread mapping.

3.2 Data Sharing Detection

LAPT requires the addition of an entry in the page table called *Sharers Vector* (SV) to identify the threads that access a page. Each SV stores the IDs of the last threads that accessed the corresponding page. Whenever SV gets full, an old thread ID needs to be removed to make room for the new one, keeping temporal locality. LAPT also introduces a *Sharing Matrix* (SM) in main memory for each parallel application. It stores an estimation of the amount of data shared between each pair of threads. Special registers containing the memory address and dimensions of SM, as well as the ID of the thread being executed, must be added to the architecture and updated by the operating system.

The behavior of LAPT is as follows. When thread T tries to access a page P but its entry is not present in the TLB, the processor performs a page table walk. Besides fetching the page table entry of P, the processor also fetches the corresponding Sharers Vector SV_P. LAPT then increments the Sharing Matrix SM in row T, for all the columns that correspond to a thread in SV_P:

$$SM[T][\ SV_P[i]\ ] \leftarrow SM[T][\ SV_P[i]\ ] + 1, \text{where } 0 \leq i < |SV_P| \tag{3.1}$$

After updating SM, LAPT inserts thread T into SV_P:

$$SV_P[i] \leftarrow SV_P[i-1], \text{where } 0 < i < |SV_P| \tag{3.2}$$

$$SV_P[0] \leftarrow T \tag{3.3}$$

3.3 Thread Mapping Computation

The information provided by the data sharing detection is used to calculate an optimized mapping of threads to cores during the execution of the parallel application. As the mapping problem is NP-hard (BOKHARI, 1981), the use of efficient heuristics are required to calculate the mapping. Online mapping requires algorithms with a short execution time, since its overhead directly affects the executing application.

To calculate the thread mapping, the operating system applies a mapping algorithm in the Sharing Matrix. In this work, we used the EagerMap (CRUZ et al., 2015) mapping algorithm, which receives the Sharing Matrix and a graph representing the memory hierarchy as input, and

it outputs which core will execute each thread. This information is then used to migrate threads to their assigned cores. Other libraries and algorithms, such as METIS (KARYPIS; KUMAR, 1998), Zoltan (DEVINE et al., 2006), Scotch (PELLEGRINI, 1994) and TreeMatch (JEANNOT; MERCIER; TESSIER, 2014), could be used. EagerMap was chosen because it is more appropriate for online mapping.

The operating system chooses the frequency in which the thread mapping routine is called. To prevent unnecessary migrations and to reduce the overhead, we adjust this frequency dynamically. If the computed mapping does not differ from the previous mapping, we increase the mapping interval by 50ms because the mapping is stable. If mappings differ, we halve the interval. The interval is kept between 50ms and 500ms to limit the overhead while still being able to react quickly to changes of data sharing behavior. The initial time interval was set to 300ms in the experiments.

3.4 Data Mapping Computation

To calculate the data mapping, the operating system verifies the contents of the Sharers Vector SV of each page in the page table. We also introduce a field in the page table to store an estimation of the amount of memory accesses from each NUMA node, which we call *NUMA Vector* (NV). This field is updated by the operating system and it has one counter per NUMA node.

When the data mapping routine is called, for every page P of the application, the counters of the NUMA nodes of each thread in SV_P are incremented in the NUMA vector of this page (NV_P). This is illustrated in Equation 3.4, where $node(x)$ is a function that returns the NUMA node in which thread x is running.

$$NV_P[\, node(SV_P[i]) \,] \leftarrow NV_P[\, node(SV_P[i]) \,] + 1, \text{where } 0 \leq i < |SV_P| \tag{3.4}$$

After the contents of NV_P are updated, we use Equations 3.5 and 3.6 to determine if P, which currently resides in node M, should be migrated to the NUMA node returned by the function $Dmap(P)$, which returns the node that performs most accesses to the page.

$$Dmap(P) = \{N \mid NV_P[N] = max(NV_P)\} \tag{3.5}$$

$$Migrate(P) = \begin{cases} true & \text{if } NV_P[Dmap(P)] \geq 2 \cdot avg(NV_P) \text{ and } Dmap(P) \neq M \\ false & \text{otherwise} \end{cases} \tag{3.6}$$

A page migration will happen only if the value of the counter of the node returned by $Dmap$ is greater or equal to twice its average value. In this way, we reduce the possibility of having

a ping-pong effect of page migrations between NUMA nodes. Naturally, a migration happens only if the node returned by the $Dmap$ function is different from the NUMA node in which page P currently resides. Also, every time a page is migrated and the average value of NV_P is at least 1, we use an aging technique in which all elements of NV are halved, making it possible to adapt to changes in access behavior.

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

3.5 Example of the Operation of LAPT

To illustrate how LAPT works, consider the example shown in Figure 3.2, where an application with 6 threads is executing on a NUMA machine with 4 nodes, and the Sharers Vector support up to 4 thread IDs. Thread 3 tries to access page P but it does not have an entry in the TLB. The processor then performs a page table walk to read the corresponding page table entry. Besides reading the physical address and page protection information, it reads the Sharers Vector, which contains thread IDs 0, 2, 1, and 4, in the order from MRU to LRU position. The core running thread 3 continues its execution. In parallel to that, LAPT increments the Sharing Matrix in cells $(3, 0)$, $(3, 2)$, $(3, 1)$ and $(3, 4)$. After that, LAPT updates the Sharers Vector, moving thread 3 to the MRU position and shifting all the other threads in SV towards the LRU position, removing thread 4.

During run time, the operating system evaluates the Sharing Matrix to update the thread

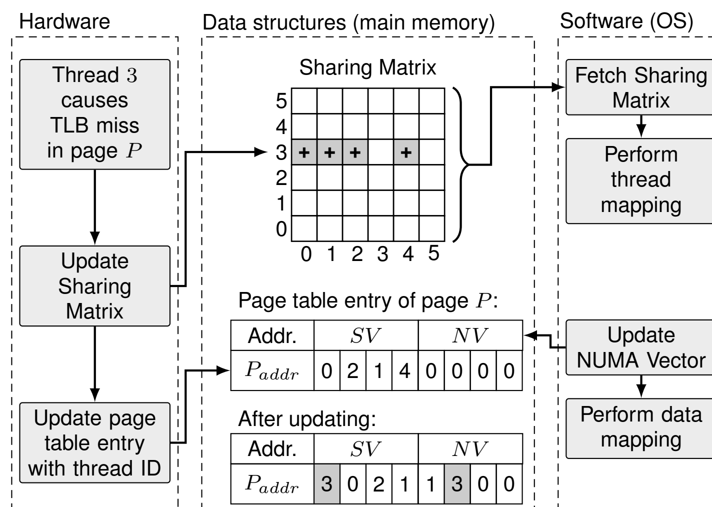


Figure 3.2: Detailed description of the LAPT mechanism. In this example, thread 3 generates a TLB miss in page P.

mapping. It changes the mapping of the threads to execute thread 0 on NUMA node 0, and migrates the other threads to node 1. Then, it evaluates the NUMA Vector of P for data mapping. The initial value of all the elements in NV_P is 0. Since only thread 0 is executing on node 0, the corresponding entry in NV_P will be incremented by 1. Likewise, the other three threads that are in SV_P are running on node 1, whose value will be incremented by 3. The node with the highest value in NV_P is node 1 (value 3) and the average of the values of NV_P is 1. Therefore, page P is migrated to node 1 following Equation 3.6.

3.6 Overhead

LAPT's overhead consists of storage space in main memory, circuit area to implement LAPT in the processor, and a run time overhead.

3.6.1 Memory Storage Overhead

The Sharing Matrix requires $4 \cdot N^2$ bytes, where N is the number of threads, considering an element size of 4 Bytes. For 256 threads, the Sharing Matrix requires 256 KByte. If the parallel application creates more threads than the maximum supported by the Sharing Matrix during run time, the operating system can allocate a larger matrix and copy the values from the old matrix.

In the last level page table entries, we store the Sharers Vector (SV) and NUMA Vector (NV). A common size for each page table entry in current architectures is 8 Bytes, as in x86-64 (INTEL, 2012a). We extend the size of each entry to 16 Bytes. We reserve 1 Byte to store each thread ID of SV, and track 4 threads per page, supporting up to 256 threads per parallel application. We also reserve 1 Byte to store each count of NV, supporting 4 NUMA nodes. The space used in the page table represent 0.2% of memory space overhead compared to the original system when using a standard 4 KByte page size. To support large systems, we would just need to use more memory.

3.6.2 Circuit Area Overhead

LAPT requires the addition of some registers, adders, and multiplexers to each core. More specifically, 64-bit registers are used to store the position in memory of SM, intermediary values to compute the memory position of an entry on SM, and the displacement to read SV for a page. 16-bit registers store the ID of the current running thread, the IDs stored on SV and one of these IDs to compute the address of an entry on SM. One 32-bit register is required to store the value of $SM[T][\ SV_P[i]\ ]$. Two 64-bit carry look-ahead adders are employed to compute the addresses of SV_P and $SM[T][\ SV_P[i]\ ]$, while a 32-bit carry look-ahead adder is used to increase the value of $SM[T][\ SV_P[i]\ ]$. Finally, multiplexers define which values are summed. In total, LAPT requires less than $25,000$ additional transistors per core, which

represents an increase in transistors of less than 0.009% in a current processor.

3.6.3 Runtime Overhead

The additional hardware introduced by LAPT is not in the critical path, since it operates in parallel to the normal operation of the processor. On the hardware level, the time overhead introduced by LAPT consists of the additional memory accesses to update the page table entries and the Sharing Matrix. The amount of additional memory accesses depends on the TLB miss rate, which differs for each application. On the software level, the time overhead includes the calculation of the thread and data mappings, and the respective migrations. The scalability of LAPT is affected very little by the number of NUMA nodes or cores. If for any reason the number of nodes or cores increases to a point that affects LAPT's overhead, the operating system could compute the mapping less frequently.

4 SAMMU – SHARING-AWARE MEMORY MANAGEMENT UNIT

Our second proposal is a mechanism called *Sharing-Aware Memory Management Unit* (SAMMU). SAMMU adds sharing-awareness to the MMU to optimize the memory accesses of parallel applications. It works in the same way for multicore and multithreaded architectures, so we will only refer to cores in the text. We begin this chapter with an overview of the concepts of SAMMU and the data structures introduced by it. Afterwards, we explain how SAMMU gathers information on memory accesses and detects the sharing pattern between the threads and the page usage pattern, and present an example of its operation. Finally, we discuss implementation details and the overhead.

4.1 Overview of SAMMU

A high-level overview of the operation of the MMU, TLB and SAMMU is illustrated in Figure 4.1. On every memory access, the MMU checks if the page has a valid entry in the TLB. If it does, the virtual address is translated to a physical address and the memory access is performed. If the entry is not in the TLB, the MMU performs a page table walk and caches the entry in the TLB before proceeding with the address translation and memory access. The operation of the MMU is extended in two ways, both happening in parallel to the normal operation of the MMU without stalling application execution:

1. SAMMU counts the number of times that each TLB entry is accessed from the local core. This enables the collection of information about the pages accessed by each thread. We

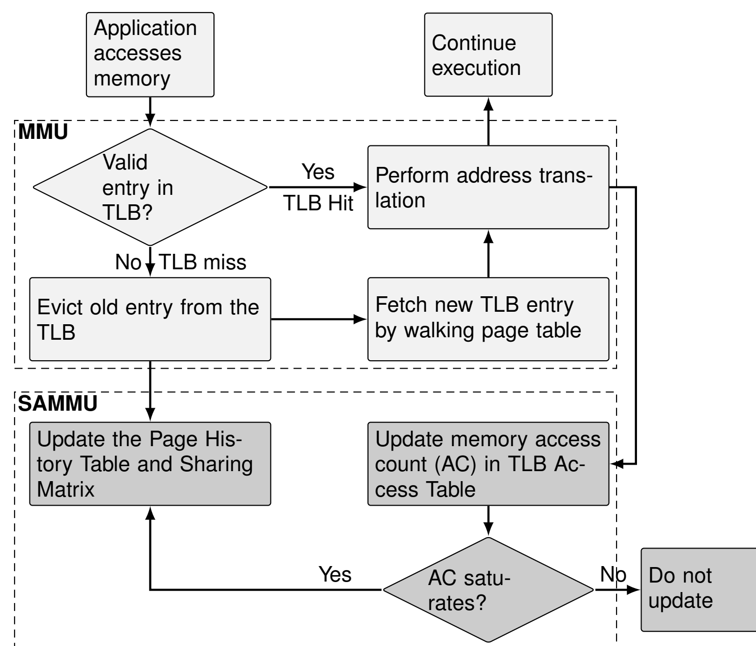


Figure 4.1: Overview of the MMU and SAMMU.

store one counter per TLB entry, which we call Access Counter (AC), in a table that we call *TLB Access Table*, which is stored in the MMU.

2. On every TLB eviction or when an Access Counter saturates, SAMMU analyzes statistics about the page and stores them in the main memory in two separate structures. The first structure is the *Sharing Matrix (SM)*, which estimates the amount of sharing between the threads. The second structure is the *Page History Table*, which contains one entry per physical page in the system with information about the threads and NUMA nodes that accessed them, and is indexed by the physical page address. Each entry of the Page History Table has three fields: (i) *Access Threshold* (AT), which defines the minimum number of memory accesses required to modify the statistics; (ii) *Sharers Vector* (SV), which contains the ID of the last threads that accessed the page; (iii) *NUMA Vector* (NV), which estimates the number of accesses from each NUMA node.

4.2 Structures Introduced by SAMMU

SAMMU adds new control registers to the architecture, and stores control data in the main memory. The following structures were added to the main memory:

Page History Table (PHT) Contains memory access information related to each page. It stores 3 fields for each page: the *Sharers Vector (SV)*, the *NUMA Vector (NV)* and the *Access Threshold (AT)*. The Sharers Vector holds the identifier of the last threads that accessed each page. The NUMA Vector holds the affinity of each NUMA node to the page. The Access Threshold specify the minimum amount of memory access a thread must perform to a page to allow it to update the Sharers Vector and NUMA Vector.

Sharing Matrix (SM) Stores the affinity between each pair of threads. The size of of the Sharing Matrix is $nt \times nt$, where nt is the number of threads per parallel application supported by SAMMU. This structure is used by the operating system to calculate the thread mapping.

SAMMU also adds an structure in the MMU, the **TLB Access Table (TAT)**. For each TLB entry cached on all TLBs, it stores the *Access Counter (AC)*, which counts the amount of memory accesses performed to the page. The TLB Access Table also stores the ID of the thread that generated the TLB miss, kept by the operating system in a control register CR_{tid}.

4.3 Gathering Information about Memory Accesses

SAMMU gathers memory access information by counting the number of memory accesses to each page in the TLB of each core. We count the number of accesses to the TLB entry of a page by adding a saturating *Access Counter* (AC) to the TLB Access Table. When AC saturates or a TLB entry gets evicted, SAMMU collects the information and updates the Page History

Table entry of the page. To filter out threads that perform only few accesses to a page, we use an *Access Threshold* (AT) in the Page History Table. AT specifies the minimum number of memory accesses required to update the mapping-related information for a page. A number of memory accesses smaller than AT means that a thread does not use a page enough to influence its mapping. SAMMU updates the mapping-related statistics of a page only when its AC saturates or if the page is evicted from a TLB and the number of memory accesses registered in the AC of this TLB entry is greater than or equal to the AT of the page.

A detailed example of the operation of SAMMU can be found in Figure 4.4. The initial value of AT is 0. The Access Threshold is kept per page because the number of memory accesses can vary from page to page. SAMMU automatically adjusts the Access Threshold of a given page, separating this procedure into two cases (Figure 4.4-Ⓑ):

Case 1 (AC saturates or $AC \geq AT$): When AC saturates or, during a TLB eviction, AC is greater than or equal to its Access Threshold (AT) (Figure 4.4-Ⓑ,Ⓓ), AT is updated with the average value of AC and AT, as illustrated in Equation 4.1. Also, the mapping statistics are updated, as explained in Sections 4.4 and 4.5. It is important to note that, since we use the same number of bits to store AC and AT, when AC saturates, it will be greater than or equal to AT.

$$AT_{new} = \frac{AT_{old} + AC}{2}, \quad AC \geq AT_{old} \tag{4.1}$$

Case 2 ($AC < AT$): In the second case, when the number of memory accesses registered by AC during a TLB eviction is lower than the Access Threshold (Figure 4.4-Ⓑ,Ⓒ), we update AT in such a way that NUMA nodes with a small number of accesses to the page have a lower influence on the threshold. Therefore, we need to find an equation $f(AC)$ that subtracts a value from AT_{old}:

$$AT_{new} = AT_{old} - f(AC) \tag{4.2}$$

Equation $f(AC)$ must have three properties (we consider AT_{old} as a constant):

$f(0) = 0$, which means that if the amount of accesses AC were 0, the Access Threshold would not be changed. The reason is that, if the thread performs no memory access to a page, we do not want this thread to influence the statistics of that page.

$f(AT_{old}) = 0$, which means that if the thread performed the same amount of access as the value of the Access Threshold, we also do not want to change its value.

$f(AC) = k$, where $0 < AC < AT_{old}$ and $0 < k < AT_{old}$. This means that, for all values of AC between 0 and AT_{old}, we want to reduce the value of the Access Threshold, but keeping its value higher than 0.

No linear equation can provide the properties required by function $f(AC)$. We then choose to use a quadratic equation. We could use other polynomial equations, but since we need to implement this in hardware, we want to use the simplest equation possible.

The quadratic equation is concave down. Therefore, to find equation $f(AC)$:

$$f(AC) = (-1) \times (AC) \times (AC - AT_{old}) \times c \tag{4.3}$$
$$= AC \times AT_{old} \times c - AC^2 \times c \tag{4.4}$$

The roots of the equation are 0 and AT_{old}. We multiply by -1 so the equation is concave down. We also multiply by a constant c to help us control the concavity of the function to keep $f(AC) < AT_{old}$. To find the possible values of c, we need to derivate $f(AC)$ to find its critical point.

$$f'(AC) = AT_{old} \times c - 2 \times c \times AC \tag{4.5}$$

To find the critical point, we equal the derivate to 0.

$$f'(AC) = 0 \tag{4.6}$$
$$AT_{old} \times c - 2 \times c \times AC = 0 \tag{4.7}$$
$$2 \times c \times AC = AT_{old} \times c \tag{4.8}$$
$$AC = \frac{AT_{old} \times c}{2 \times c} \tag{4.9}$$
$$AC = \frac{AT_{old}}{2} \tag{4.10}$$

Now, we know that, regardless the value of c and AT_{old}, the maximum value of $f(AC)$ happens when $AC = {}^{AT_{old}}/{}_2$. That is, when the Access Counter is half of the current Access Threshold. We can use this value of AC in Equation 4.4 to get the maximum value that Equation $f(AC)$ can subtract from AT_{old} in Equation 4.2.

$$f\left(\frac{AT_{old}}{2}\right) = \frac{AT_{old}}{2} \times AT_{old} \times c - \left(\frac{AT_{old}}{2}\right)^2 \times c \tag{4.11}$$
$$= \frac{AT_{old}^2}{2} \times c - \frac{AT_{old}^2}{4} \times c \tag{4.12}$$
$$= \frac{2}{4} \times AT_{old}^2 \times c - \frac{AT_{old}^2}{4} \times c \tag{4.13}$$
$$= \frac{c \times AT_{old}^2(2-1)}{4} \tag{4.14}$$
$$= \frac{c \times AT_{old}^2}{4} \tag{4.15}$$

Equation 4.15 specifies the maximum value of Equation $f(AC)$. We need now to find the

possible values of c such that $f(AC) < AT_{old}$.

$$\frac{c \times AT_{old}^2}{4} < AT_{old} \tag{4.16}$$

$$c < \frac{4 \times AT_{old}}{AT_{old}^2} \tag{4.17}$$

$$c < \frac{4}{AT_{old}} \tag{4.18}$$

Since we want to reduce the impact of threads that access the page few times as much as possible, we chose to use $c = {}^{1}/_{AT_{old}}$. Therefore, the equation used to update the Access Threshold in **Case 2** is given by Equation 4.23, found by substituting the value of c in Equations 4.2 and 4.4 by $c = {}^{1}/_{AT_{old}}$.

$$AT_{new} = AT_{old} - f(AC) \tag{4.19}$$

$$= AT_{old} - \left(AC \times AT_{old} \times c - AC^2 \times c\right) \tag{4.20}$$

$$= AT_{old} - \left(AC \times AT_{old} \times \frac{1}{AT_{old}} - AC^2 \times \frac{1}{AT_{old}}\right) \tag{4.21}$$

$$= AT_{old} - \frac{AC \times (AT_{old} - AC)}{AT_{old}} \tag{4.22}$$

$$AT_{new} = AT_{old} - \frac{AT_{old} - AC}{\left(\frac{AT_{old}}{AC}\right)}, \quad AC < AT_{old} \tag{4.23}$$

Equation 4.23 guarantees that AT will never be decreased by more than 25% at each update. In this case, mapping statistics are not updated. Further details on how we implement Equation 4.23 in hardware are given in Section 4.7.1.

The behavior of the equations that control the updates of the Access Threshold are illustrated in Figures 4.2 and 4.3. In the following equations, we prove that AT will never be decreased by more than 25% at each update. For that, the value of ${}^{AT_{new}}/_{AT_{old}}$ must be greater than or equal to 0.75.

$$\frac{AT_{new}}{AT_{old}} = \frac{AT_{old} - \frac{AT_{old} - AC}{\left(\frac{AT_{old}}{AC}\right)}}{AT_{old}} \tag{4.24}$$

$$= 1 - \frac{AT_{old} - AC}{\left(\frac{AT_{old}^2}{AC}\right)} \tag{4.25}$$

$$= 1 - \frac{AC \times AT_{old} - AC^2}{AT_{old}^2} \tag{4.26}$$

At this point, we can make a change of variables, considering $AC = \alpha AT_{old}$. Since in

Case 2 $AC < AT$, α must be $0 \leq \alpha < 1$.

$$\frac{AT_{new}}{AT_{old}} = 1 - \frac{\alpha AT_{old} \times AT_{old} - (\alpha AT_{old})^2}{AT_{old}^2} \tag{4.27}$$

$$= 1 - \frac{\alpha AT_{old}^2 - \alpha^2 AT_{old}^2}{AT_{old}^2} \tag{4.28}$$

$$= 1 - \left(\alpha - \alpha^2\right) \tag{4.29}$$

$$= \alpha^2 - \alpha + 1 \tag{4.30}$$

Therefore, we know that $^{AT_{new}}/_{AT_{old}}$ is a quadratic equation and is concave up, which means it has a global minimum value. To get the minimum value of AT_{new} relative to AT_{old}, we need first to derivate Equation 4.30.

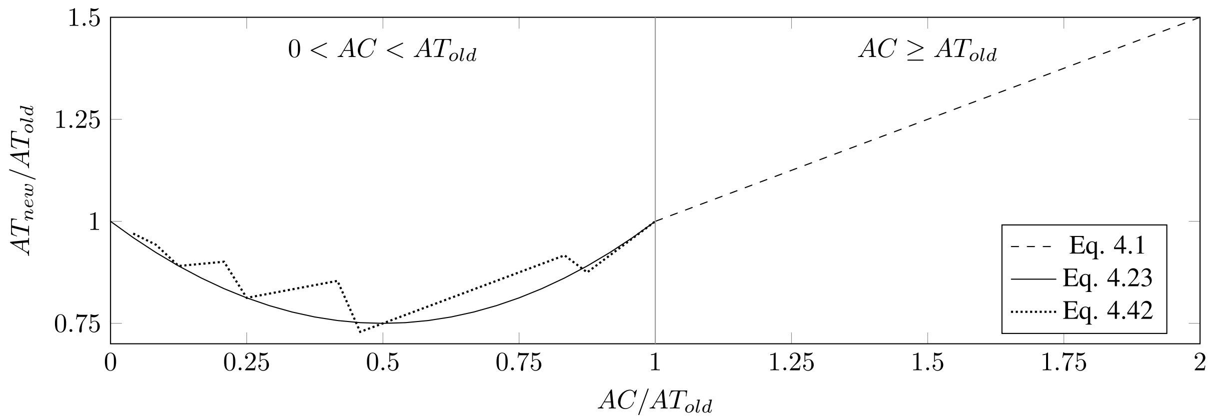
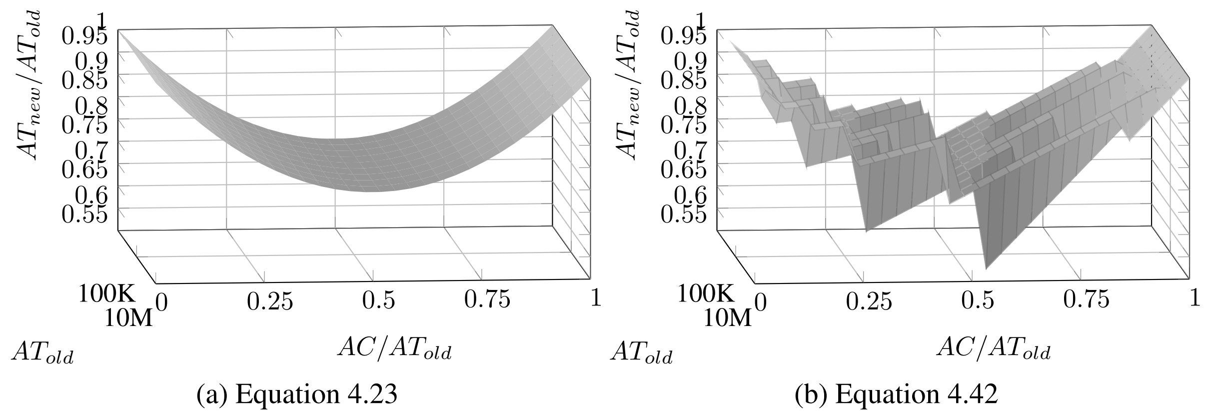


Figure 4.2: Value of AT_{new} relative to AT_{old}. For Equation 4.42, we consider AT_{old} as $10M$.



(a) Equation 4.23

(b) Equation 4.42

Figure 4.3: Behavior of Equations 4.23 and 4.42 varying AT_{old} from $100K$ to $10M$.

$$\frac{d(\alpha^2 - \alpha + 1)}{d\alpha} = 2\alpha - 1 \tag{4.31}$$

Now, we need to find the value of α when the derivate is equal to 0.

$$2\alpha - 1 = 0 \tag{4.32}$$

$$\alpha = \frac{1}{2} \tag{4.33}$$

Then, we know that the minimum value of AT_{new} relative to AT_{old} happens when $\alpha = 1/2$. In other words, when the value of the Access Counter AC is half of the Access Threshold AT. To find the minimum value, we need to insert the value of $\alpha = 1/2$ in Equation 4.30.

$$\frac{AT_{new}}{AT_{old}} = \alpha^2 - \alpha + 1 \tag{4.34}$$

$$= \left(\frac{1}{2}\right)^2 - \frac{1}{2} + 1 \tag{4.35}$$

$$= \frac{3}{4} = 0.75 \tag{4.36}$$

With this, we demonstrated that, for **Case 2**, the minimum value of AT_{new} is $0.75 \times AT_{old}$, a 25% decrease, and happens when AC is half of the AT.

4.4 Detecting the Sharing Pattern for Thread Mapping

To detect the sharing pattern, SAMMU identifies the last threads that accessed a memory page. To obtain that information, SAMMU adds a small *Sharers Vector* (SV) to each Page History Table entry. Each SV stores the IDs of the last threads to access its page. This has the advantage of maintaining temporal locality when detecting which threads share each page. Old entries will be overwritten and not considered as sharers. SAMMU also keeps a *Sharing Matrix* (SM) in main memory for each parallel application to estimate the number of accesses to pages that are shared between each pair of threads. In the TLB Access Table, SAMMU stores the ID of the thread that accessed each TLB entry. Control registers containing the memory address and dimensions of the Sharing Matrix, and the ID of the thread being executed must be added to the architecture and updated by the operating system.

When SAMMU is triggered for a certain page by thread T (Figure 4.4-Ⓐ), it accesses the SV of the corresponding Page History Table entry. If the Access Counter is greater than or equal to the Access Threshold (Figure 4.4-Ⓑ), SAMMU then increments the Sharing Matrix in

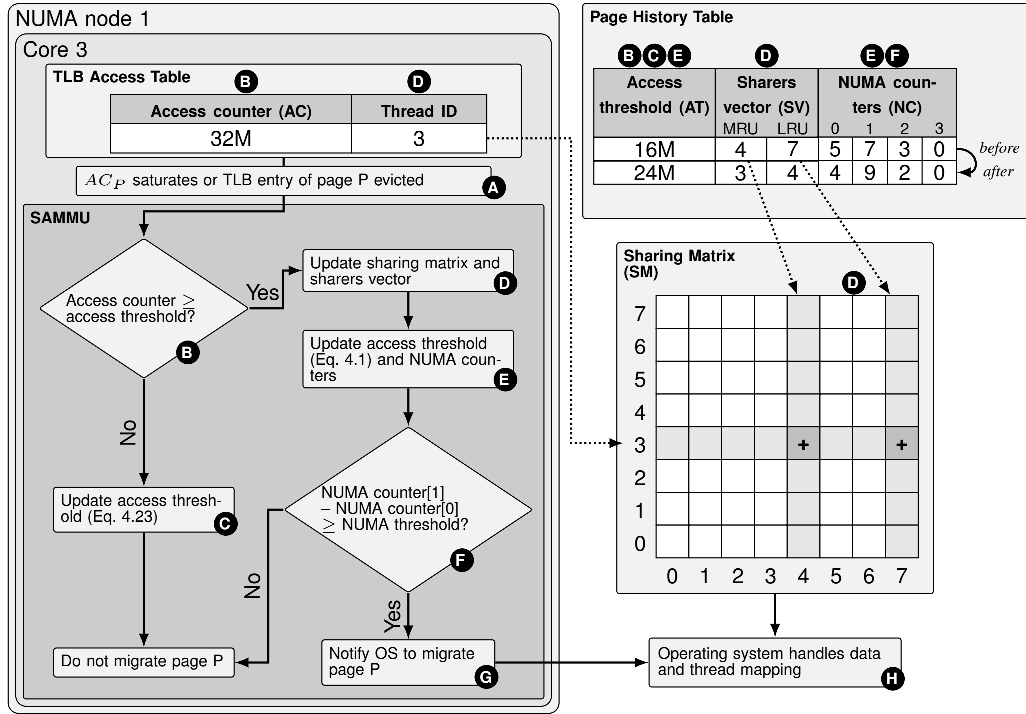


Figure 4.4: Operation of SAMMU. Structures that were added to the normal operation of the hardware and operating system are marked in **bold**. The system consists of 4 NUMA nodes with 2 cores each. The NUMA Threshold is 4. Page P is initially located on NUMA node 0. In this example, thread 3 (core 3, NUMA node 1) saturates the AC of page P.

row T, for all the columns that correspond to an entry in the SV (Figure 4.4-D).

$$SM[T][SV[i]] = SM[T][SV[i]] + 1 \tag{4.37}$$

Each line of SM is accessed by its corresponding thread only, minimizing the impact of coherence protocols. Finally, SAMMU inserts thread T into the SV of the evicted page by shifting its elements, such that the oldest entry is removed.

$$SV[i] \leftarrow SV[i-1], \qquad i \geq 1 \tag{4.38}$$

$$SV[0] \leftarrow T \tag{4.39}$$

4.5 Detecting the Page Usage Pattern for Data Mapping

To identify where a memory page should be mapped, SAMMU requires the addition of a vector to each Page History Table entry. The vector, which we call *NUMA Vector* (NV), has N counters for a system with N NUMA nodes. NV employs saturating counters to count a relative number of accesses from different NUMA nodes. The initial value of each NV is 0.

When a TLB entry from a core in NUMA node n is selected for eviction or its AC reaches its maximum value (Figure 4.4-Ⓐ), SAMMU reads the corresponding Page History Table entry. If the number of memory accesses stored in AC is greater than or equal to the threshold AT (Figure 4.4-Ⓑ), SAMMU increments the NUMA Vector counter of node n by V_{add}, and decrements all other counters by 1 (Figure 4.4-Ⓔ), as in Equation 4.40. V_{add} is a control register configured by the operating system, and its value must be at least 2. Since the NUMA Vector counters are saturated, they do not overflow nor underflow.

$$NV[x] = \begin{cases} NV[x] + V_{add} & \text{if } x = n \\ NV[x] - 1 & \text{otherwise} \end{cases} \tag{4.40}$$

After updating the values of NV, SAMMU checks if the corresponding page is stored in NUMA node n. If the page is currently mapped to another NUMA node m, SAMMU evaluates if the difference between the NUMA Vector counters of n and m is greater than or equal to a global value *NUMA Threshold* (NT) (Figure 4.4-Ⓕ), as in Equation 4.41. If that is the case, SAMMU notifies the operating system of the page and its destination node n (Figure 4.4-Ⓖ). The NUMA Threshold may be configured by a control register.

$$\textit{NotifyOS} = \begin{cases} true & \text{if } NV[n] - NV[m] \geq NT \\ false & \text{otherwise} \end{cases} \tag{4.41}$$

The operating system then chooses how it will handle the migration of the page (Figure 4.4-Ⓗ). The higher the NUMA Threshold NT, the lower the number of page migrations.

To demonstrate why V_{add} must be at least 2, consider the situation where nodes n and m present a similar access pattern to a page P, the value of V_{add} is 1, and all NV are set to 0. Nodes other than n and m do not access page P. Whenever a core from node n evicts page P (or AC of page P saturates), the value of $NV[n]$ would be incremented to 1 and $NV[m]$ would be decremented to 0. Likewise, whenever a core from node m evicts the same page P, the value of $NV[m]$ would be incremented to 1 and $NV[n]$ would be decremented to 0. In this scenario, the mechanism would not be able to detect that nodes n and m present more accesses than the others. However, if the value of V_{add} is 2, $NV[n]$ and $NV[m]$ would present higher values after multiple accesses. Therefore, by incrementing with a higher number, SAMMU is able to handle this access behavior. If more than two NUMA nodes access a page in similar amounts, a value higher than 2 is required.

4.6 Example of the Operation of SAMMU

In the situation illustrated in Figure 4.4, there are 8 cores, 4 NUMA nodes, the NUMA Threshold NT is 4, the Sharers Vector SV has 2 positions and V_{add} is 2. AC and AT support

values up to 32M. SAMMU was configured to support up to 8 threads per parallel application. Consider that page P is initially located on NUMA node 0. If thread 3, which is being executed on core 3 (NUMA node 1), saturates the AC of page P (Figure 4.4-Ⓐ), SAMMU accesses the Page History Table entry of page P.

Since page P was accessed 32M times and its AC saturated, AT (16M) will be updated to 24M following Equation 4.1 (Figure 4.4-Ⓑ,Ⓔ). For the sharing pattern, SAMMU checks the Sharers Vector SV, and finds the IDs of threads 4 and 7. It will then increment cells $(3, 4)$ and $(3, 7)$ of the Sharing Matrix by 1, and store thread 3 in SV (Figure 4.4-Ⓓ). This will result in the removal of thread 7 from SV.

Regarding the page usage pattern (Figure 4.4-Ⓔ), the NUMA Vector counter of node 1 will be incremented by 2, and all others will be decremented by 1. The counter of node 3 remains as 0 because the counter is saturated. Since the difference between $NV[1]$ (node of core 3) and $NV[0]$ (current node that stores page P) is greater than or equal to the NUMA Threshold (4) (Figure 4.4-Ⓕ), SAMMU notifies the operating system to migrate page P to node 1 (Figure 4.4-Ⓖ,Ⓗ).

Supposing that page P was accessed 4M times instead of 32M, and was selected for a TLB eviction, no updates to the Sharing Matrix and NUMA Vector would be performed, since AC would be lower than the Access Threshold (16M). In this case, the Access Threshold would be updated to 13M following Equation 4.23 (Figure 4.4-Ⓒ).

4.7 Implementation Details

We discuss several aspects of implementing SAMMU.

4.7.1 Hardware Implementation

SAMMU can be implemented in several ways. We implemented it in a way to handle only one event at a time to keep a simpler hardware. If a TLB eviction happens or an AC saturates while SAMMU is already handling another event, we disconsider this new event. Since these events are very frequent, we do not expect a significant impact on accuracy. We evaluate this in Section 6.6.

To count the number of accesses to each TLB entry, we added the TLB Access Table to the MMU. In order to implement the TLB Access Table as a scratchpad, without the need for tag comparison, we added to each TLB entry a static unique identifier. When a TLB entry is accessed, its identifier is returned along with the page information. SAMMU then uses this identifier as index in the TLB Access Table.

Regarding the updates on the AT in the Page History Table, since Equation 4.23 requires division operations, we update the Access Threshold using the approximation shown in Equation 4.42 instead, where $\gg$ represents the bit shift operation. The H function returns the posi-

tion of the highest bit set to 1 in its parameter. The behavior of this equation is also shown in Figures 4.2 and 4.3b.

$$AT_{new} = AT - \big[\big(AT - AC\big) \gg \big(H(AT) - H(AC)\big)\big] \tag{4.42}$$

4.7.2 Notifying the Operating System About Page Migrations

To notify the operating system when a page needs to migrate, SAMMU can either introduce an interruption or save in memory a list of pages to be migrated and their destination NUMA nodes. To avoid interrupting the operating system too frequently, we chose the latter. The operating system periodically checks this list and performs the migrations.

4.7.3 Performing the Thread Mapping

How often the thread mapping is performed depends on the operating system, as SAMMU is responsible only for detecting the patterns. A simple implementation consists of analyzing the Sharing Matrix and then mapping threads from time to time. To calculate the thread mapping, the operating system applies a mapping algorithm in the Sharing Matrix. As in LAPT, we also used the EagerMap (CRUZ et al., 2015) mapping algorithm, which receives the Sharing Matrix and a graph representing the memory hierarchy as input, and it outputs which core will execute each thread. We check for thread migrations every 100ms. To reduce the influence of old values, we apply an aging technique in the Sharing Matrix every time the mapping mechanism is called by multiplying all of its elements by 0.75. We evaluated values between 0.6 and 0.95, but the results were not sensitive to this value.

4.7.4 Increasing the Supported Number of Threads

The operating system starts an application configuring SAMMU to support a certain number of threads using a control register. If the parallel application creates more threads than the maximum supported, the operating system can change this during execution. To do that, it must allocate a new Sharing Matrix SM and copy the values from the old SM. It must also update the contents of the Sharers Vector SV of all pages to use the new number of bits per entry. Since this is an expensive procedure, we recommend to avoid it by configuring SAMMU to support a large number of threads from the beginning. For all systems and applications evaluated, configuring SAMMU to support 256 threads was enough to avoid this procedure.

4.7.5 Handling Context Switches and TLB Shootdowns

In context switches, if the context of a core changes to another process and thereby to another memory address space, the TLB may be flushed depending on the architecture and operating system. In case the TLB gets flushed, SAMMU needs to flush the TLB Access Table. In case the TLB is not flushed, SAMMU still detects the sharing correctly, since it stores the thread ID in the TLB Access Table. The content stored in the Page History Table is not affected. When individual TLB entries are flushed by TLB shootdowns due to changes on its page table entry, the flushed TLB entry can be tracked by SAMMU. Both context switches and changes on page table entries are much less frequent than TLB evictions and have no significant impact on the accuracy of SAMMU.

4.8 Overhead

The overhead consists of storage space in the main memory, circuit area to implement SAMMU, and execution time.

4.8.1 Memory Storage Overhead

The Page History Table and Sharing Matrix are stored in the main memory. For the configuration shown in Table 6.1, each entry of the Page History Table would require 8 Bytes. The Page History Table space overhead would be 0.2% relative to the total main memory. The Sharing Matrix would require 256 KByte, each of its elements with 4 Bytes. In this configuration, SAMMU can track up to 256 threads per parallel application. To support larger systems, we would need only to use more space per Page History Table entry, allocating more NUMA Vector counters and Sharers Vector entries, and a larger Sharing Matrix.

4.8.2 Circuit Area Overhead

The logic of SAMMU is implemented in the MMU of each core. In total it requires 8 adders, 2 subtractors, 5 shifters, and 7 multiplexers of various sizes (from 4 to 32 bits) per core. SAMMU also uses a TLB Access Table that stores in SRAM one Access Counter and one thread ID per TLB entry. The number of bits of each Access Counter must be enough to store the maximum possible value of the Access Threshold. In this scenario, the additional hardware required by SAMMU represents $143,000$ transistors per core, which results in an increase in transistors of less than 0.05% in a modern processor.

4.8.3 Execution Time Overhead

The additional hardware of SAMMU is not in the critical path, since it operates in parallel to the MMU, such that application execution is not stalled while SAMMU is operating. Therefore, the time overhead introduced by SAMMU consists of the additional memory accesses to update the Page History Table and Sharing Matrix, which depend on the TLB miss rate. To keep the overhead of these memory accesses low, SAMMU does not lock any structure before its update. The Sharing Matrix does not need to be locked since each row is updated only by one thread. For a Page History Table entry, a race condition can happen in case threads evict the same page (or the corresponding ACs saturate) at the same time. This race condition would not cause the application to fail, just a slight reduction in accuracy. Since the time SAMMU takes to update a Page History Table entry is small, this race condition is a rare event. On the software level, the operating system introduces overhead when calculating the thread mapping, and when migrating threads and pages. The measured execution time overhead from both hardware and software are shown in Section 6.6.

5 IPM – INTENSE PAGES MAPPING

Our third proposal is a mechanism called *Intense Pages Mapping* (IPM). In architectures with hardware-managed TLBs, IPM is implemented as an addition to the memory management unit (MMU) hardware, because the MMU has direct access to the TLB and all memory accesses. In architectures with software-managed TLBs, IPM can be implemented natively in the TLB miss handler of the operating system. This chapter is organized as follows. It first analyses the usage of TLB time as a metric for sharing-aware mapping. Then, it presents an overview of IPM, followed by a discussion on the additional data structures required to capture memory access patterns. A detailed description of IPM is given next, and an account of its overhead completes the chapter.

5.1 Detecting Memory Access Patterns via TLB Time

We intend to improve the accuracy of sharing-aware mapping with a novel method to detect memory access patterns and data sharing based on the time a page table entry stays cached in the TLB. We refer to it as **TLB time**. We show in this section that this metric estimates the whole memory access behavior of an application with higher accuracy than others. This happens because intensely accessed pages tend to stay for longer times in the TLBs of the cores that make these accesses. Additionally, this metric provides a smaller cost to implement in hardware than using the number of memory accesses itself, as the latter requires the addition of counters to every TLB entry (TIKIR; HOLLINGSWORTH, 2008), as in SAMMU.

Following the hypothesis that a more accurate estimation of the memory access pattern of an application can result in a better mapping and performance improvements, we performed experiments using the TLB time, memory access sampling, and TLB misses as metrics to guide thread and data mapping. Experiments were run using the NAS parallel benchmarks (JIN; FRUMKIN; YAN, 1999) and the PARSEC benchmark suite (BIENIA et al., 2008) on a two NUMA node machine with software-managed TLB (more information in Section 6.1.3). This

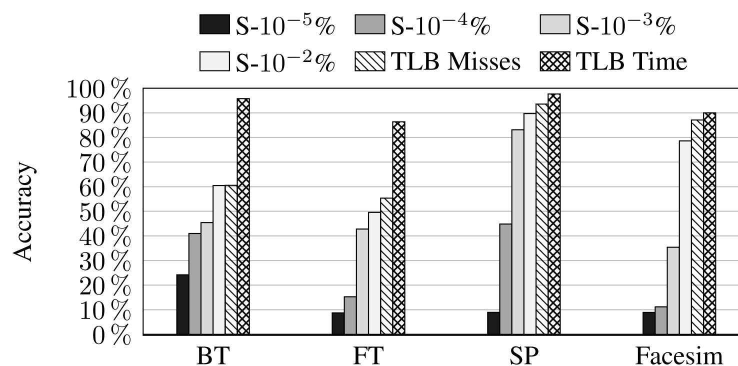


Figure 5.1: Accuracy results obtained with different metrics for sharing-aware mapping.

machine was used because it provides high accuracy information about TLB misses. TLB time and TLB misses were captured directly from the TLB miss handler in the operating system, while memory accesses were sampled using Pin (BACH et al., 2010). We use all memory accesses as a baseline for accuracy, and we vary the number of samples from $10^{-5}\%$ to $10^{-2}\%$ of the whole for comparison. As stated in Section 1.2, a realistic sampling based mechanism (DIENER et al., 2015a) uses at most $10^{-3}\%$ of samples to avoid harming performance, and another sampling based mechanism (DASHTI et al., 2013) specify that it samples once every $130,000$ cycles.

The accuracy obtained with the different methods for benchmarks BT, FT, SP, and Facesim are presented in Fig. 5.1. To calculate the accuracy, we compare if the NUMA node selected for each page of the applications is equal to the NUMA node that performed most accesses to the page (the higher the percentage, the better). The execution time is calculated as the reduction compared to using $10^{-5}\%$ samples (the bigger the reduction, the better). The accuracy results show that the TLB time provides estimations that are more accurate than all other tested methods. The differences are most noticeable with the BT and FT benchmarks, where the TLB time shows over 85% of accuracy, while the other methods are all under 60%. One may also notice that accuracy increases with the number of memory access samples used, as expected. Nevertheless, most sampling mechanisms are limited to a small number of samples due to the high overhead of the techniques used for capturing memory accesses (AZIMI et al., 2009; DIENER et al., 2014), harming accuracy.

The results indicate that accurate information about the memory access behavior is important for a good mapping. Furthermore, as we showed in Section 1.2, a good mapping leads to better performance improvements. Our proposed metric provided the best accuracy, and has almost the same implementation complexity as the TLB misses metric. Given the benefits of using the TLB time over other methods, the next sections present a mechanism for capturing the TLB time and employing it for online data and thread mapping.

5.2 Overview of IPM

A high-level overview of the operation of the MMU, TLB and IPM is illustrated in Figure 5.2. On every memory access, the MMU checks if the page has a valid entry in the TLB. If it does, the virtual address is translated to a physical address and the memory access is performed. If the entry is not in the TLB, the MMU performs a page table walk and caches the entry in the TLB before proceeding with the address translation and memory access. IPM modifies the behavior of the MMU during a TLB miss.

On every TLB miss, IPM stores a time stamp in the main memory in a structure called *TLB Access Table*. If a TLB entry must be evicted to store the new entry, IPM loads from the *TLB Access Table* the time stamp corresponding to the evicted TLB entry. By subtracting the loaded time stamp from the current time stamp, it knows how much time the evicted entry stayed

cached in the TLB. IPM then uses this time difference to update memory sharing information in two structures: the *Page History Table* and *Sharing Matrix*. Finally, IPM notifies the operating system if a page migration could improve performance.

5.3 Structures Introduced by IPM

IPM adds new control registers to the architecture, and stores control data in the main memory. The following structures were added to the main memory:

Page History Table (PHT) Contains memory access information related to each page. It stores 2 fields for each page: the Sharers Vector *(SV)* and the NUMA Vector *(NV)*. The Sharers Vector holds the identifier of the last threads that accessed each page. The NUMA Vector holds the affinity of each NUMA node to the page.

TLB Access Table (TAT) For each TLB entry cached on all TLBs, it stores the time stamp related to the moment when the corresponding TLB entry was fetched on a page table walk. It is used to compute the TLB time used for mapping. The TLB Access Table also stores the ID of the thread that generated the TLB miss, kept by the operating system in a control register CR_{tid}.

Sharing Matrix (SM) Stores the affinity between each pair of threads. The size of of the Sharing Matrix is $nt \times nt$, where nt is the number of threads per parallel application supported by IPM. This structure is used by the operating system to calculate the thread

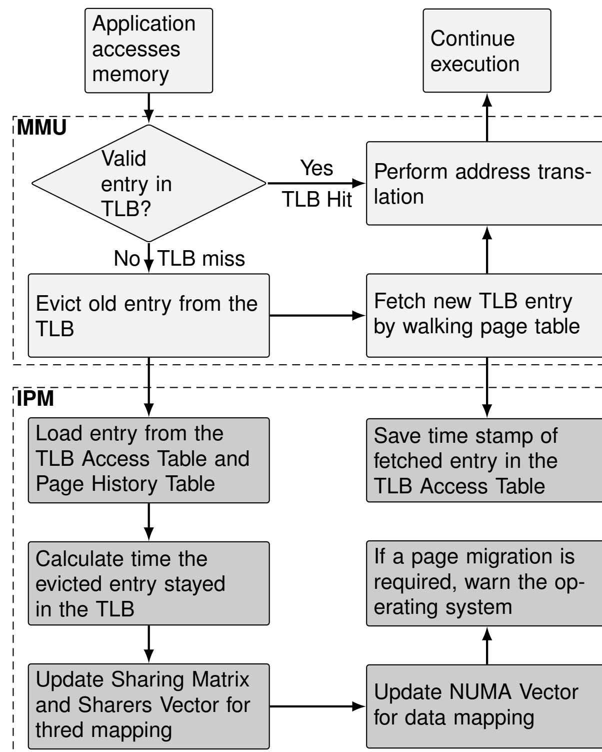


Figure 5.2: Overview of the MMU and IPM.

mapping.

Since each TLB entry has a corresponding entry in the TLB Access Table, IPM adds a unique static identifier to each TLB entry. This identifier is hardwired, not requiring any SRAM. The identifier is returned along with the physical address when the TLB is accessed, and is used as an index into the TLB Access Table (further details in Section 5.4.1).

5.4 Detailed Description of IPM

IPM is responsible for three tasks: computing the TLB time for different page entries and threads; detecting the page sharing for thread mapping; and discovering the memory affinity of pages to NUMA nodes for data mapping.

5.4.1 Calculating the TLB Time

The time spent in the TLB is the main information required by IPM to analyze the memory access pattern. For this end, the architecture must provide a counter that is incremented at a constant rate over time. Most architectures already provide this counter, such as the x86-64 architecture with its *time stamp counter* (TSC), which can be read using the `rdtsc` instruction. Whenever a TLB miss happens and it requires the eviction of an older entry, IPM fetches from the TLB Access Table (TAT) the entry corresponding to the evicted page, and saves in registers the elapsed time and thread ID. This is shown in Equations 5.1 and 5.2, where *TLBEntryID* is the ID of the TLB entry introduced in Section 5.3.

$$Elapsed \leftarrow (TSC \gg CR_{shift}) - \mathit{TAT}[\mathit{TLBEntryID}].ts \tag{5.1}$$

$$\mathit{Tid} \leftarrow \mathit{TAT}[\mathit{TLBEntryID}].tid \tag{5.2}$$

Since the time stamp has a clock cycle precision, we can discard the lowest bits of the elapsed time with a negligible accuracy loss. The operating system can configure the number of bits to be discarded using the CR_{shift} control register. After that, since the TLB Access Table entry of the evicted page is already stored in registers, IPM overwrites the TLB Access Table entry with the current time stamp, also shifted by CR_{shift}, and the thread ID of the current thread running in the core, indicated in the CR_{tid} control register.

5.4.2 Gathering Information for Thread Mapping

To perform thread mapping, we need to know the affinity between the threads. Threads with higher affinity should be mapped to cores close together in the memory hierarchy. To detect the

affinity, IPM keeps the last threads that accessed a memory page in the Sharers Vector *(SV)* of the Page History Table. Whenever a TLB entry of page P is evicted, the corresponding Page History Table entry is read. After that, the Sharing Matrix *(SM)* is incremented by the elapsed time (Equation 5.1) in row Tid (Equation 5.2), for all the columns that correspond to an entry in the SV_P, as shown in Equation 5.3.

$$SM[Tid][SV_P[i]] \leftarrow SM[Tid][SV_P[i]] + Elapsed \tag{5.3}$$

Each line of the Sharing Matrix is accessed by its corresponding thread only, minimizing the impact of coherence protocols. Finally, IPM inserts thread Tid (Equation 5.2) into the SV of the evicted page P by shifting its elements, removing its oldest entry.

$$SV_P[i] \leftarrow SV_P[i-1], \qquad i \geq 1 \tag{5.4}$$

$$SV_P[0] \leftarrow Tid \tag{5.5}$$

To calculate the thread mapping, the operating system applies a mapping algorithm in the Sharing Matrix. As in our other proposals, we also used the EagerMap (CRUZ et al., 2015) mapping algorithm, which receives the Sharing Matrix and a graph representing the memory hierarchy as input, and it outputs which core will execute each thread. In our experiments, this is done every 200ms. In order to reduce the influence of old values in the Sharing Matrix, we apply an aging technique every time the mapping mechanism is called by multiplying all of its elements by 0.75. Values between 0.6 and 0.95 were also evaluated, but the results showed no sensitivity to this value.

5.4.3 Gathering Information for Data Mapping

To perform data mapping, the operating system needs to know the affinity of each memory page to each NUMA node. In this way, the operating system can map pages to the nodes that most access them. To achieve this, IPM introduces the NUMA Vector *(NV)* in each entry of the Page History Table. The NUMA Vector has one counter per NUMA node. Each counter estimates the affinity of the corresponding page to each node. When a TLB entry corresponding to a page P is evicted and its Page History Table entry is loaded, IPM performs aging in each counter of the NUMA Vector, as in Equation 5.6. Aging is important to detect changes in the access pattern to a page. The higher the value of the CR_{aging} control register, less aging is performed.

$$NV_P[i] \leftarrow NV_P[i] - (NV_P[i] \gg CR_{aging}) \tag{5.6}$$

After aging, the NUMA Vector element related to the NUMA node where the thread is running is incremented by the elapsed time (Equation 5.1) of the evicted TLB entry, shown in

Equation 5.7. In this equation, n represents the NUMA node where the thread that generated the TLB miss is running. It is important to note that the counters of the NUMA Vector are saturating counters, such that they keep their maximum possible value in case of overflow.

$$NV_P[n] \leftarrow NV_P[n] + elapsed \tag{5.7}$$

After incrementing the counters, IPM evaluates if the evicted page is intensely used by a specific node and should be migrated. This is shown in Equation 5.8, where n is the node of the core executing the thread, m is the node that currently stores page P, and CR_{mig} is a control register to help decide whether a migration is necessary. We left shift $NV_P[m]$ by CR_{mig} to avoid migrating pages that are accessed by several nodes. The higher the value of CR_{mig}, the more difficult it is to migrate a page. If the condition of Equation 5.8 is true, then the page is intensely used by a node and IPM notifies the operating system to migrate page P to node n.

$$\textit{NotifyOS} = \begin{cases} true & \text{if } NV_P[n] > (NV_P[m] \ll CR_{mig}) \\ false & \text{otherwise} \end{cases} \tag{5.8}$$

It is important to mention that the initial value of each counter of the NUMA Vector should not be zero. A zero value would result in too many migrations early during the execution. The initial value was defined as in Equation 5.9, which is the maximum value not affected by aging, and is set by the operating system when initializing the page table entry.

$$NV_P[i] = (1 \ll CR_{aging}) - 1 \tag{5.9}$$

5.5 Overhead

We implemented IPM in a way to handle only one TLB eviction at a time to keep the hardware modifications limited. If a TLB eviction happens while IPM is already handling another eviction, we do not consider this new eviction. Since TLB evictions are very frequent, we do not expect a significant impact on accuracy even when discarding a fraction of them, which is evaluated in Section 6.6. The overhead of IPM consists of storage space in the main memory, circuit area, and a performance overhead since it operates during application execution. We calculated the overhead with the configuration shown in Table 6.1.

5.5.1 Memory Storage Overhead

The TLB Access Table, Page History Table and Sharing Matrix are stored in the main memory. Each entry of the TLB Access Table requires 8 bytes: 7 bytes to store the shifted time stamp counter, and 1 byte to store the thread ID. Each entry of the Page History Table would require 16 Bytes (we need to roundup the size of the Page History Table entry to a power-of-

two). The Page History Table space overhead would be 0.4% relative to the total main memory using a 4 KBytes page size. The Sharing Matrix would require 256 KByte, each of its elements with 4 Bytes. In this configuration, IPM can track up to 256 threads per parallel application, supports 4 NUMA nodes, and there are 4 unused bytes. To support larger systems, we would need only to use these unused bytes, or use more space per Page History Table entry, allocating more NUMA Vector and Sharers Vector entries, and a larger Sharing Matrix.

5.5.2 Circuit Area Overhead

In architectures with hardware-managed TLBs, IPM is implemented in the MMU of each core. It requires the addition of some registers, carry look-ahead adders and subtractors, shifters, and multiplexers. 64 bit registers are used to store the positions in memory of the Sharing Matrix, Page History Table and TLB Access Table, while control registers and others store values using 16 bits or less.

An implementation of IPM optimized for area and power (by reusing some of the circuits for the different equations described previously) requires less than $30,000$ additional transistors per core, which represents an increase of less than 0.02% in a current processor. Additionally, a performance-oriented implementation of IPM (with replicated resources) can be done by using less than double of the number of transistors of the other implementation.

5.5.3 Execution Time Overhead

The additional hardware of IPM is not in the critical path, since it operates in parallel to the MMU, such that application execution is not stalled while IPM is operating. Therefore, the time overhead introduced by IPM consists of the additional memory accesses to update the TLB Access Table, Page History Table and Sharing Matrix, which depend on the TLB miss rate. To keep the overhead of these memory accesses low, IPM does not lock any structure before its update. Any possible race condition would not cause the application to fail, just a slight reduction in accuracy. On the software level, the operating system introduces overhead when calculating the thread mapping, and when migrating threads and pages. The measured execution time overhead from both hardware and software are shown in Section 6.6.

6 EVALUATION OF OUR PROPOSALS

In this chapter, we analyze and perform several experiments with our proposed mechanisms. We describe the methodology adopted, the accuracy of our detection mechanisms and how our mechanisms improve the performance and energy consumption. We also analyze the overhead of our proposals, and how they perform when varying several configuration parameters.

6.1 Methodology

In this section, we describe the experiments we performed to evaluate our proposals, including the environments and workloads employed. We used 3 types of machines to analyze our proposal in different scenarios. A full system simulator is used to evaluate the operation in machines with hardware-managed TLBs. We used a real machine with a software-managed TLB to prove that IPM (and our other proposals) can work not only in a simulator, but also in real machines. We performed a trace-based evaluation using real machines with hardware-managed TLBs to gain precise information regarding how our proposals affect the performance, cache misses, interchip traffic, and energy consumption in a real architecture, with more reliable results than in the simulator. Table 6.1 summarizes the parameters used.

6.1.1 Algorithm to Map Threads to Cores

In all of our proposals, we used the EagerMap (CRUZ et al., 2015) mapping algorithm to generate the thread mapping, which receives the Sharing Matrix and a graph representing the memory hierarchy as input, and it outputs which core will execute each thread. This information is then used to migrate threads to their assigned cores. Other libraries and algorithms, such as METIS (KARYPIS; KUMAR, 1998), Zoltan (DEVINE et al., 2006), Scotch (PELLEGRINI, 1994) and TreeMatch (JEANNOT; MERCIER; TESSIER, 2014), could be used. As previously stated, EagerMap is more appropriate for online mapping. A detailed description of EagerMap, as well as a comparison to other mapping algorithms, is given in Appendix A.

6.1.2 Evaluation Inside a Full System Simulator

We implemented our proposals in the Simics simulator (MAGNUSSON et al., 2002), extended with the GEMS-Ruby memory model (MARTIN et al., 2005) and the Garnet interconnection model (AGARWAL et al., 2009). The simulated machine runs Linux and has 4 processors, each with 2 cores, with private L1 caches, and L2 caches shared by all cores. Each processor is on a different NUMA node. Cache latencies were calculated using CACTI (THOZIYOOR et al., 2008), and the memory timings from JEDEC (JEDEC, 2012). Both the intrachip and interchip interconnection topologies are bidirectional rings. We simulate the benchmarks with

Table 6.1: Configuration of the experiments.

	Parameter	Value
SAMMU	Structure sizes	AC, AT: 32 bits, SV: 2x 8 bits, NV: 4x 4 bits
	Sharing matrix	256 threads, 4 Byte element size
	Control registers	Support up to 256 threads, $V_{add} = 2$, $NT = 10$
IPM	Structure sizes	*SV*: 2x 8 bits, *NV*: 4x 16 bits
	Sharing matrix	256 threads, 4 Byte element size
	Control registers	CR_{shift} : 13, CR_{aging} : 7, CR_{mig} : 2
Pin	L1 TLB	64 entries, 4-way, shared between 2 SMT-cores
	L2 TLB	512 entries, 4-way, shared between 2 SMT-cores
Itanium	Processors	4x Intel Itanium 9030 (Montecito), 2 cores
	Caches/proc.	2x 16 KByte L1, 2x 256 KByte L2, 2x 4 MByte L3
	Main memory	2 NUMA nodes, 16 GByte DDR-400, 16 KByte page size
	Environment	Linux kernel 2.6.32, GCC 4.4
Xeon32	Processors	2x Xeon E5-2650 (SandyBridge), 8 cores, 2-SMT
	Caches/proc.	8x 32 KByte L1, 8x 256 KByte L2, 20 MByte L3
	Main memory	2 NUMA nodes, 32 GByte DDR3-1600, 4 KByte page size
	Environment	Linux kernel 3.8, GCC 4.6
Xeon64	Processors	4x Xeon X7550 (Nehalem), 8 cores, 2-SMT
	Caches/proc.	8x 32 KByte L1, 8x 256 KByte L2, 18 MByte L3
	Main memory	4 NUMA nodes, 128 GByte DDR3-1333, 4 KByte page size
	Environment	Linux kernel 3.8, GCC 4.6
Simics	Processors	4x 2 cores, 2.0 GHz, 32 nm
	L1 cache/proc.	2x 16 KByte, 4-way, 1 bank, 2 cycles latency
	L2 cache/proc.	1 MByte, 8-way, 2 banks, 5 cycles latency
	TLB/proc.	2x TLBs (64 entries), 4-way, 1 TLB per core
	Cache coherency	Directory-based MOESI protocol, 64 Byte lines
	Main memory	8 GByte DDR3-1333 9-9-9, 4 KByte page size
	Interconnection	1/40 cycles latency (intra/interchip) 64/16 Byte bandwidth (intra/interchip)
	Environment	Linux kernel 2.6.15, GCC 4.3

small input sizes due to simulation time constraints. To compensate for the small input sizes, we reduced the size of caches memories and TLBs accordingly, as done in (CUESTA et al., 2013).

In Simics, we execute each test only once due to the simulation time. We compare our proposals to the following mapping policies:

Default (baseline) The default thread mapping policy is the Linux scheduler. The default data mapping policy is controlled by GEMS, which is interleaving.

Oracle The oracle mapping is generated by tracking all memory accesses in the simulator to keep a Sharing Matrix and a NUMA Vector. Every 100ms, threads are migrated using the mapping provided by EagerMap, and pages are migrated to the nodes that most access them.

6.1.3 Evaluation Inside Real Machines with Software-Managed TLBs

Although our proposals are extensions to the current hardware of MMUs, it is possible to implement the behaviors of LAPT and IPM in software in architectures that have a software-managed TLB. In these architectures, whenever a TLB miss happens, there is no hardware page table walker to fetch the page table entry from the main memory. Instead, the architecture generates a TLB miss interrupt that is handled by the operating system, which walks through the page table in software. The Itanium architecture (INTEL, 2010a) has a hybrid mechanism to handle TLB misses, where we can choose to use a hardware-based manager called Virtual Hash Page Table (VHPT) walker, or handle the TLB misses in software.

We implemented IPM in the Linux kernel version 2.6.32 for the Itanium architecture configured to use a software-managed TLB. We refer to this machine as *Itanium*. The architecture also provides access to a cycle accurate time stamp by reading the `ar.itc` register. The code that performs the same procedure of IPM was added to the data TLB miss interrupt handler of the kernel, which is directly implemented in Assembly. In this handler, we keep the Sharing Matrix and a list of memory pages candidate for migration and the destination node. In a kernel module, we create a kernel thread that verifies these structures every 100ms and performs the migrations.

In Itanium, we execute each test 10 times, and show a 95% confidence interval calculated with the Student's t-distribution. We use this machine to compare our proposals (although we implemented only IPM in Itanium) to the related work because it is a real machine and we can run every mechanisms, except the oracle mapping, natively. We compare our proposals to the following mapping policies:

Operating system (baseline) The thread mapping policy is the default Linux scheduler, with a first-touch data mapping.

Interleave We interleave the memory pages across the NUMA nodes. The thread mapping policy is the default Linux scheduler.

Oracle The oracle mapping is generated by tracking all memory accesses. It is similar to the oracle used in Simics, but using Pin (BACH et al., 2010) to monitor all memory accesses, since the real machine does not provide such information. Then, when running the application, the oracle mapping is fed to a kernel module we developed that maps the threads and data.

NUMA Balancing The NUMA Balancing policy (CORBET, 2012b) was included in more recent versions of the Linux kernel. In this policy, the kernel introduces page faults during the execution of the application, and pages are migrated to the node that generated the page fault. The thread mapping policy is the default Linux scheduler. NUMA Balancing is not implemented in the kernel running in Itanium, such that we had to port its code to allow us to evaluate its performance.

kMAF kMAF (DIENER et al., 2014) also works using page faults, but also performs sharing-aware thread mapping and keeps a history regarding page migrations to reduce ping ponging of pages between nodes.

TLB misses Similarly to NUMA Balancing, we migrate pages to the NUMA node that generated a TLB miss on that page.

TLB misses – IPM Does the same procedure of IPM, keeping a Sharing Matrix, Sharers Vector and a NUMA Vector, but using TLB misses instead of TLB time.

6.1.4 Trace-Based evaluation Inside Real Machines with Hardware-Managed TLBs

The experiments were performed using two different real machines. The first machine, *Xeon32*, consists of two NUMA nodes with one Intel Xeon E5-2650 processor per node, with a total of 32 virtual cores. The second machine, *Xeon64*, consists of four NUMA nodes with one Intel Xeon X7550 processor per node, with a total of 64 virtual cores. The machines are running version 3.8 of the Linux kernel. Information about the hardware topology is gathered using Hwloc (BROQUEDIS et al., 2010b).

Since our proposals are extensions to the current MMU hardware, we simulate their behavior with Pin (BACH et al., 2010). Pin is a platform to develop instrumentation tools through callback routines. We used Pin for the analysis because it is faster than a full system simulator. It analyzes the binary code and identifies the places where the analysis code needs to be inserted, which is done by a just-in-time compiler. The analysis code is provided by the programmer of the instrumentation tool. In our instrumentation tool that simulates our proposals, we instrumented Pin to monitor all memory accesses and the creation and destruction of threads. In the tool, we simulate the behavior of our proposals and the basic functions of the TLB. We do not simulate more hardware components to keep simulation speed higher than the one of Simics.

The simulated TLB in Pin uses the same TLB topology as the real machines, with two TLB levels. The L2 TLB is inclusive. To make it possible to evaluate it in real machines, the information about the thread and data mappings generated in Pin are fed into the mapping mechanism in runtime. To keep the execution inside the simulator synchronized with the execution in the real machine, we keep track of the barriers of the applications in the simulator. This is possible in applications that always execute the same amount of barriers and in the same order along different executions. Therefore, when the execution in the simulator and real machine reach the same barrier, their executions are in the same state.

In the real machine, besides the performance, we measured L2 and L3 cache misses per thousand instructions (MPKI) and interchip interconnection traffic (QuickPath Interconnection) traffic. In Xeon64, these measurements were made with the Intel PCM tool (INTEL, 2012b). In Xeon32, cache misses were measured using Intel PCM, and interchip interconnection traffic was estimated using Intel VTune by measuring the amount of cache to cache transfers and remote memory accesses (Intel PCM presented issues when monitoring interchip traffic in Xeon32).

We also measured both processor and DIMM energy consumption and calculate the amount of energy per instruction as an indicator of energy efficiency. Energy consumption was measured with Intel PCM using the Running Average Power Limit (RAPL) hardware counters (INTEL, 2012a) introduced in the Intel SandyBridge architecture. We only show the energy results for Xeon32 because Xeon64 does not support RAPL.

All experiments in the real machine were executed 30 times. We show average values as well as a 95% confidence interval calculated with Student's t-distribution. We compare the results of our proposal to:

Operating system (baseline) The thread mapping policy is the default Linux scheduler, with a first-touch data mapping.

Random For the random mapping, we randomly generated a thread and data mapping for each execution.

Oracle The oracle mapping is generated by tracking all memory accesses. It is similar to the oracle used in Simics, but using Pin (BACH et al., 2010) to monitor all memory accesses, since the real machine does not provide such information. Then, when running the application, the oracle mapping is fed to a kernel module we developed that maps the threads and data.

6.1.5 Presentation of the Results

In the figures containing results comparing our proposals to related work, the results are normalized using Equation 6.1. We also show a 95% confidence interval calculated with Student's t-distribution.

$$Result(x) = \frac{Value(x) - Value(baseline)}{Value(baseline)} \tag{6.1}$$

6.1.6 Workloads

As workloads, we used the OpenMP implementation of the NAS parallel benchmarks (NPB) (JIN; FRUMKIN; YAN, 1999), v3.3.1, and the PARSEC benchmark suite (BIENIA et al., 2008), v3.0. We configured the benchmarks to run with one thread per virtual core, although some applications of PARSEC execute with multiple threads per virtual core.

For the evaluation in the real machines, the evaluated applications must present the same sharing and page usage patterns across different executions, as well as keeping the same memory address space, since the trace generated in Pin is used to guide mapping decisions. For this reason, only the NAS applications (except DC) were executed on the real machines. DC and applications from PARSEC usually do not keep the same memory address space due dynamic memory allocation. This makes the information generated in Pin unreliable to guide mapping

in future executions.

Input sizes were chosen to provide similar total execution times and feasible simulation times. Regarding NAS, benchmarks BT, LU, SP and UA were executed using input size *A* in Pin, Xeon32 and Xeon64, and input size *W* in Simics. Benchmarks CG, EP, FT, IS and MG were executed using input size *B* in Pin, Xeon32 and Xeon64, and input size *A* in Simics. DC was executed with input size *W* in Simics. Regarding PARSEC in Simics, the input size used in Canneal was *simmedium*, and all others *simlarge*. In the Itanium machine, DC was executed with input size *A* and all other NAS benchmarks with input size *B*, and the ones of PARSEC with the *native* input size.

We also experiment with a real world scientific application, Ondes3D (DUPROS et al., 2008). Ondes3D simulates the propagation of seismic waves using a finite-differences numerical method. The oracle mapping in Ondes3D was not generated by analyzing memory traces because the execution time of Ondes3D is too large to enable the generation of a trace. Therefore, the oracle mapping in Ondes3D is performed by manually adding the mapping routines in its source code. By exploiting the regular memory access pattern of finite-differences applications, we guarantee that the memory accessed by each thread is mapped nearby (DUPROS et al., 2010).

6.2 Comparison Between our Proposed Mechanisms

In Section 2.4, we defined several characteristics that are desirable for a mechanism to perform sharing-aware mapping. In this context, all of our proposals have all characteristics:

- Perform both thread and data mapping
- Support dynamic environments
- Support dynamic memory allocation
- Has access to memory addresses
- Low overhead
- Low trade-off between accuracy and overhead
- No manual source code modification
- Do not require sampling

Nevertheless, there are differences between our proposals. The main differences are:

Hardware/Software responsibilities In LAPT, the hardware is responsible just for analyzing which threads access each data, while the software must verify the NUMA nodes running the threads to check for page migrations. On the other hand, in SAMMU and IPM, the hardware already verifies the best NUMA node for each page, the software has only to perform the migrations.

Accuracy LAPT has the lowest accuracy among our proposals, since it uses information only

regarding TLB misses. On the other hand, SAMMU has information about the amount of memory accesses, stored in the Access Counter in the TLB Access Table, and IPM has access to the time each page has its entry cached in the TLB, stored in the TLB Access Table.

Overhead LAPT presents the highest overhead because the operating system needs to iterate over the entire page table to verify if any page needs to be migrated. This is not required in SAMMU and IPM.

Flexibility LAPT is more flexible because the policy to check for page migrations can be easily changed by the operating system. In SAMMU and IPM, the operating system can only configure the amount of migrations by setting the control registers.

Implementation cost LAPT has lower implementation cost, as most of its analysis is performed in the software level. Between SAMMU and LAPT, SAMMU has a higher cost because it requires one Access Counter for all entries in the TLB Access Table.

6.3 Accuracy of the Proposed Mechanisms

We analyze the accuracy of the sharing matrices and the detected data mappings.

6.3.1 Accuracy of the Sharing Patterns for Thread Mapping

The first results we show are the sharing matrices detected by our proposals, as well as the accuracy of the data mappings. We generated them using Pin, configuring the applications to create 32 threads. Regarding NAS, benchmarks BT, DC, LU, SP and UA were executed using input size *A*, and benchmarks CG, EP, FT, IS and MG were executed using input size *B*. Regarding PARSEC, the input size used was *simlarge*. The sharing matrices are shown in Figures 6.1 to 6.23. Axes represent thread IDs. Cells show the amount of accesses to shared data for each pair of threads. Darker cells indicate more accesses. We normalized the contents of each matrix to its own maximum value.

In BT (Figure 6.1), IS (Figure 6.6), SP (Figure 6.9) and UA (Figure 6.10), our proposals were able to correctly detect the data sharing between neighbor threads. The data sharing between thread 0 and all other threads, which consists in an initialization or reduction pattern, as shown in our baseline (Figure 2.11), was not detected by our proposals. However, this does not represent any issue, because the detected neighboring data sharing is more relevant regarding thread mapping. As shown in our baseline using 1 and 2 sharers (Figure 2.18), the predominant pattern is the neighboring data sharing. Also, thread mapping considering neighboring data sharing has potential for performance improvements, while the initialization and reduction patterns cannot be improved.

In MG (Figure 6.8), we were able to detect data sharing only between neighboring threads, while our baseline also indicates data sharing between distant threads (Figure 2.11h). In LU

(Figure 6.7), LAPT detected only the neighboring data sharing pattern, and SAMMU and IPM also detected data sharing between distant threads. Both patterns are correct, as in our baseline, LU presented neighboring data sharing with up to 4 thread sharers per memory block (Figures 2.19a to 2.19c), and present data sharing between both neighboring and distant threads with more sharers (Figures 2.19d to 2.19f). In Fluidanimate (Figure 6.17) and x264 (Figure 6.23), data sharing between near and distant threads were detected according to the baseline (Figure 2.11). In the applications following a pipeline parallelization model, Ferret (Figure 6.16) and Streamcluster (Figure 6.21), the detected pattern, is very similar to the baseline. Dedup (Figure 6.14) also follows a pipeline model, but the detected patterns were slightly different compared to the baseline, although the sharing clusters can be clearly seen.

Blackscholes (Figure 6.11) and Facesim (Figure 6.15) presented data sharing between neighboring threads using our proposals, which were not present in the baseline (Figure 2.11), although Facesim showed up some traces of it in our baseline with 32 KBytes pages (Figure 2.13d). In this case, when the baseline present a sharing pattern unsuitable for thread mapping, but our proposals detect another pattern, there is no problem in terms of performance. The reason for this is simple: if the performance of the application is not influenced by thread mapping, then any thread mapping we detect should not have any impact on performance. Due to this, we would only consider a detected pattern as wrong if the baseline pattern is considered as good for thread mapping. Following this rationality, although the baseline of CG (Figure 2.11b) shows a very homogeneous all-to-all pattern, and our proposals show an all-to-all pattern with a considerable amount of noise, we do not consider this as a problem. This reasoning can be extended to several other applications, more specifically DC, EP, FT, Bodytrack, Canneal, Freqmine, Raytrace, Swaptions and Vips.

6.3.2 Accuracy of the Data Mapping

Regarding data mapping, we need to analyze the accuracy of our proposals, verifying if the pages of the parallel application were mapped to the correct NUMA node. By correct NUMA node, we mean the NUMA node that performed most amount of memory accesses to the corresponding page. To calculate the accuracy, we used Equations 6.2 and 6.3. In these Equations, N_P is the total number of pages, $Access(p, n)$ is a function that returns the amount of memory accesses of NUMA node n to page p, $Access(p)$ is a function that returns the total amount of memory accesses from each node to page p, and $NodeMap(p)$ returns the node that stores page p. Pages that have more memory accesses have a higher influence on the accuracy. Figure 6.24 shows the accuracy of our proposed mechanisms, as well as the default policy of

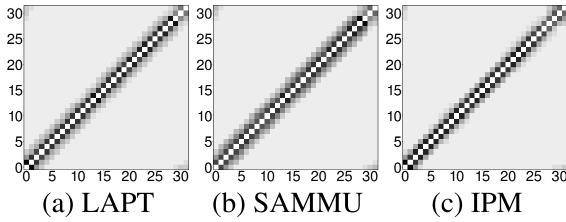

(a) LAPT (b) SAMMU (c) IPM

Figure 6.1: Sharing patterns of BT.

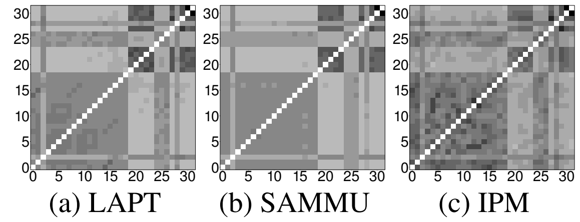

(a) LAPT (b) SAMMU (c) IPM

Figure 6.2: Sharing patterns of CG.

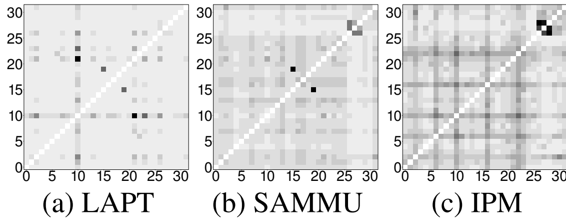

(a) LAPT (b) SAMMU (c) IPM

Figure 6.3: Sharing patterns of DC.

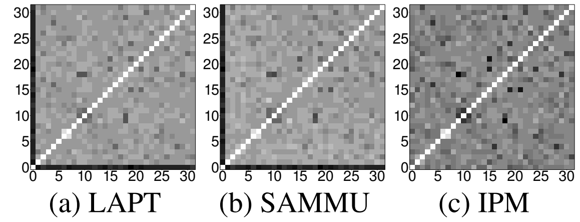

(a) LAPT (b) SAMMU (c) IPM

Figure 6.4: Sharing patterns of EP.

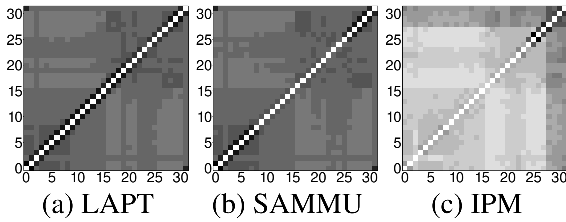

(a) LAPT (b) SAMMU (c) IPM

Figure 6.5: Sharing patterns of FT.

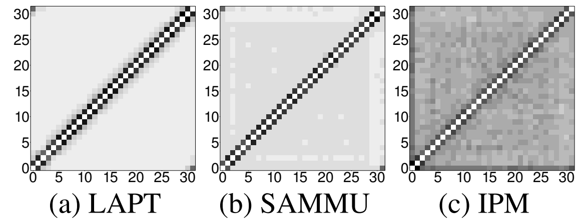

(a) LAPT (b) SAMMU (c) IPM

Figure 6.6: Sharing patterns of IS.

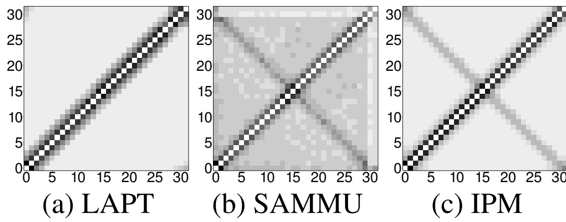

(a) LAPT (b) SAMMU (c) IPM

Figure 6.7: Sharing patterns of LU.

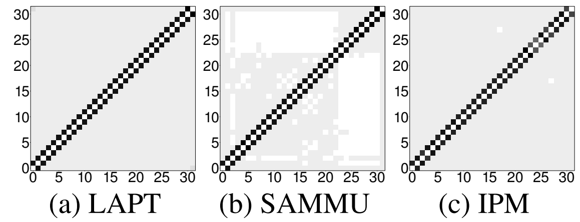

(a) LAPT (b) SAMMU (c) IPM

Figure 6.8: Sharing patterns of MG.

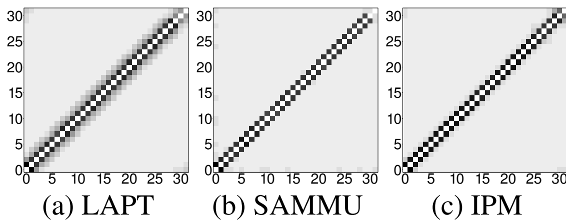

(a) LAPT (b) SAMMU (c) IPM

Figure 6.9: Sharing patterns of SP.

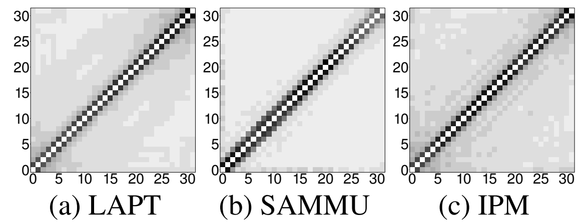

(a) LAPT (b) SAMMU (c) IPM

Figure 6.10: Sharing patterns of UA.

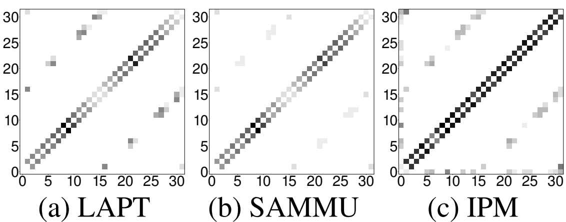

(a) LAPT (b) SAMMU (c) IPM

Figure 6.11: Sharing patterns of Blackscholes.

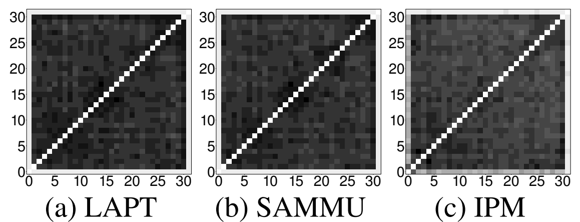

(a) LAPT (b) SAMMU (c) IPM

Figure 6.12: Sharing patterns of Bodytrack.

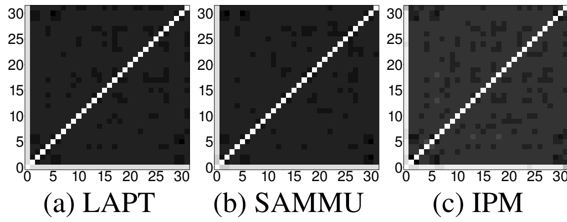

(a) LAPT (b) SAMMU (c) IPM

Figure 6.13: Sharing patterns of Canneal.

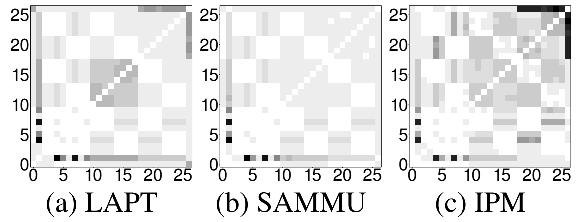

(a) LAPT (b) SAMMU (c) IPM

Figure 6.14: Sharing patterns of Dedup.

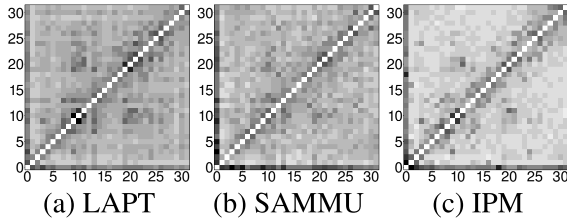

(a) LAPT (b) SAMMU (c) IPM

Figure 6.15: Sharing patterns of Facesim.

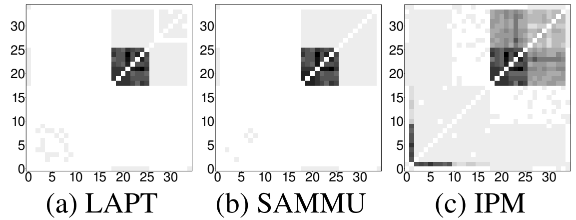

(a) LAPT (b) SAMMU (c) IPM

Figure 6.16: Sharing patterns of Ferret.

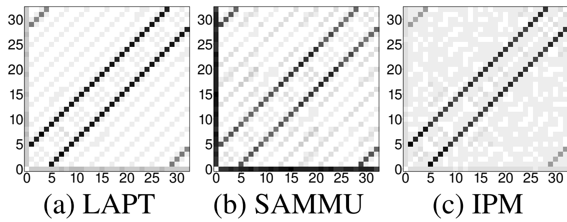

(a) LAPT (b) SAMMU (c) IPM

Figure 6.17: Sharing patterns of Fluidanimate.

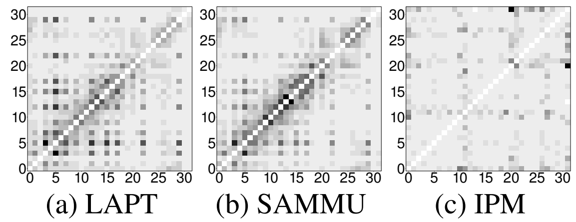

(a) LAPT (b) SAMMU (c) IPM

Figure 6.18: Sharing patterns of Freqmine.

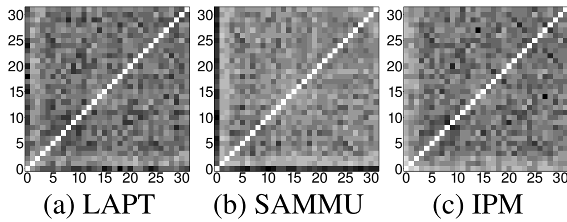

(a) LAPT (b) SAMMU (c) IPM

Figure 6.19: Sharing patterns of Raytrace.

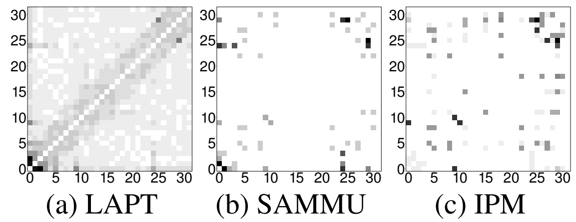

(a) LAPT (b) SAMMU (c) IPM

Figure 6.20: Sharing patterns of Swaptions.

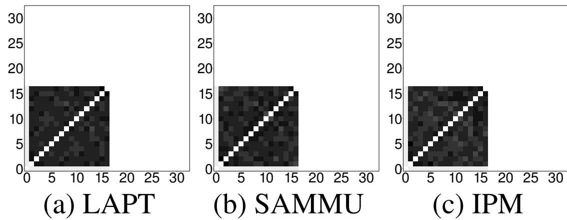

(a) LAPT (b) SAMMU (c) IPM

Figure 6.21: Sharing patterns of Streamcluster.

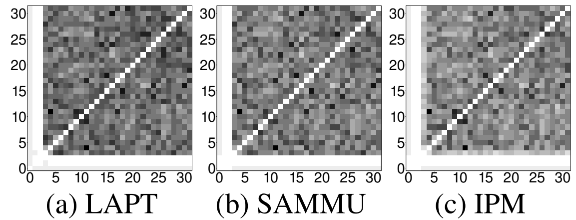

(a) LAPT (b) SAMMU (c) IPM

Figure 6.22: Sharing patterns of Vips.

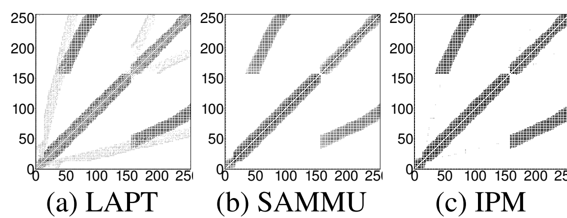

(a) LAPT (b) SAMMU (c) IPM

Figure 6.23: Sharing patterns of x264.

Linux, the first-touch policy, and the exclusivity level of each application (Section 2.1.3.2).

$$Correct(p, map) = \begin{cases} 1 & \text{if } map = node | Access(p, node) = max(Access(p)) \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

$$Accuracy = \frac{\sum_{p=1}^{N_P} Access(p) \cdot Correct(p, NodeMap(p))}{\sum_{p=1}^{N_P} Access(p)} \tag{6.3}$$

On average, the accuracy of LAPT, SAMMU and IPM were 75.7%, 79.7% and 78.7%. The average accuracy of the first-touch was 63.1%. We can observe a clear relation between the exclusivity level and the accuracy of our proposals. Ours proposals present a higher accuracy in applications that have a higher exclusivity level. The reason is that, in applications that have a higher exclusivity, the pages are more accessed by a single thread, decreasing the influence of the other threads. Thereby, finding the most appropriate NUMA node is easier in applications with higher exclusivity levels. It is also important to mention that having a lower accuracy in applications that have a low exclusivity is not a problem, since these applications do not benefit from data mapping anyway.

Comparing the average accuracy, the difference between our proposals to the first-touch policy ranges from 12.6% to 16.6%. Although the average difference to the first-touch is not very high, the difference in some applications is very significant. For instance, in BT, LAPT, SAMMU and IPM have an accuracy of 82.5%, 92.5% and 96.2%, respectively, while the accuracy of the first-touch policy was only 37.7%. This happens because the accuracy of the first-touch depends if the programmer of the parallel application initialized the data accordingly. In this context, it is important to highlight that several applications are programmed considering that operating system uses the first-touch policy to map the data (JIN; FRUMKIN;

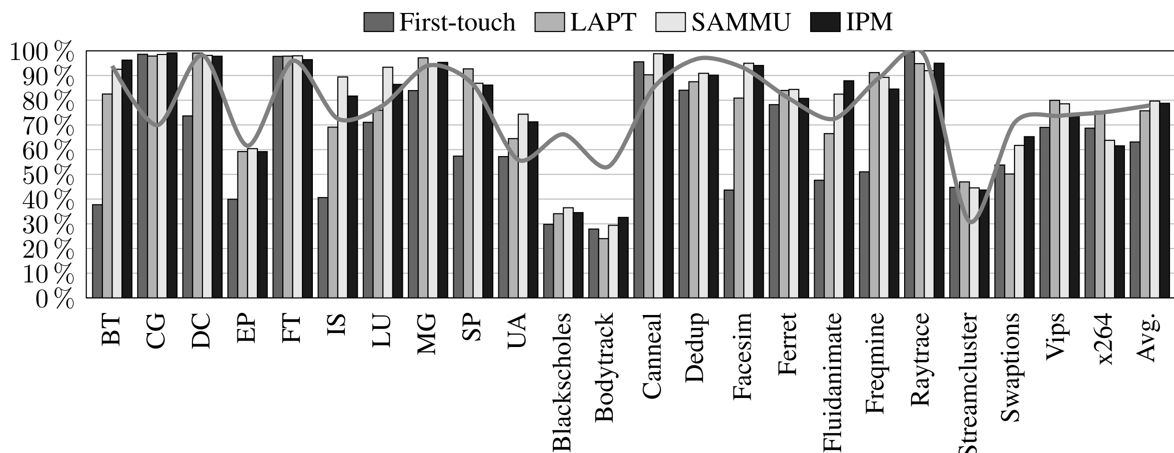


Figure 6.24: Accuracy of the data mappings using our proposed mechanisms and the first-touch policy (default policy of Linux). The gray line over the bars represents the exclusivity level of the applications (Section 2.1.3.2).

YAN, 1999), which increases the accuracy of the first-touch. However, we believe that this is not an appropriate solution because of three things:

- It imposes the burden of the data mapping to the programmer, since the programmer will have to modify the code in order to make each thread initialize their own data.
- In some applications, the optimal data mapping can change during execution, which is not supported by first-touch.
- If the operating system does not use first-touch, the data mapping will not be optimized.

6.4 Performance Experiments

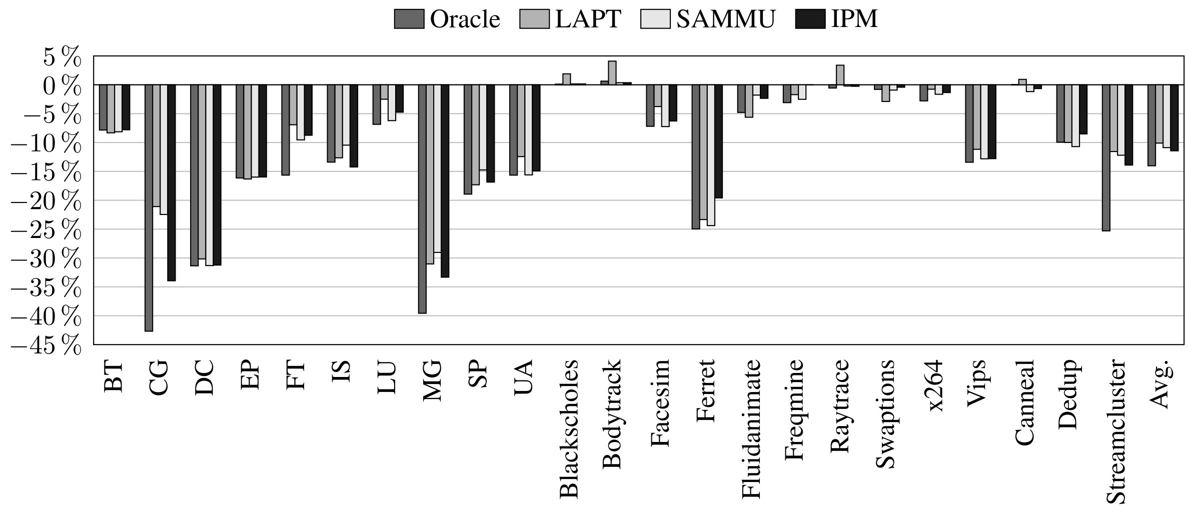
We evaluate the performance of our mechanisms in the different platforms separately.

6.4.1 Performance Results in Simics/GEMS

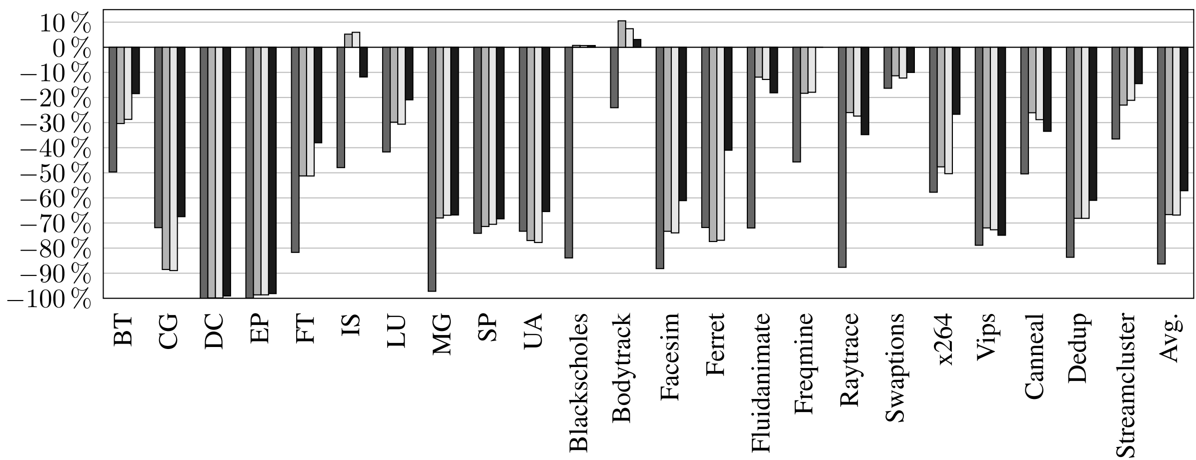
The performance results obtained in Simics are shown in Figure 6.25. We show the execution time, in Figure 6.25a, the amount of interchip interconnection traffic, in Figure 6.25b, and the amount of L2 cache misses per kilo instructions (MPKI), in Figure 6.25c. On average, execution time was reduced by 10.1%, 10.9% and 11.4% for LAPT, SAMMU and IPM, respectively. Following the same mechanisms sequence, on average, interchip interconnection traffic was reduced by 66.6%, 66.8% and 57.1%, and the L2 misses were reduced by 12.1%,12.5% and 8.9%. We can observe that the NAS applications presented better improvements than the PARSEC ones. The reason for that is because PARSEC applications are more influenced than NAS applications by factors other than memory locality, such as load balancing, that are not in the scope of this thesis.

The application in which LAPT and SAMMU presented the best improvements was MG, reducing execution time by 31.0% and 29.0%. MG, as we can see in its sharing pattern (Figure 6.8), can benefit from both thread and data mappings. Due to that, we improve interchip interconnection traffic, by 68.0% and 66.9% respectively for LAPT and SAMMU, as well as L2 misses, by 11.8% and 8.8%. Regarding IPM, the best improvements happened in CG, where execution time was reduced by 33.9%, due to a substantial reduction of interchip traffic of 67.5% and L2 misses of 25.5%. Although the sharing pattern of CG does not indicate that it is suitable for thread mapping, IPM reduced the amount of L2 misses because it performed less thread migrations than the other mechanisms. Every time a thread migrates to another core, the hardware transfers on demand the data to the cache memory of this new core, increasing the amount of cache misses.

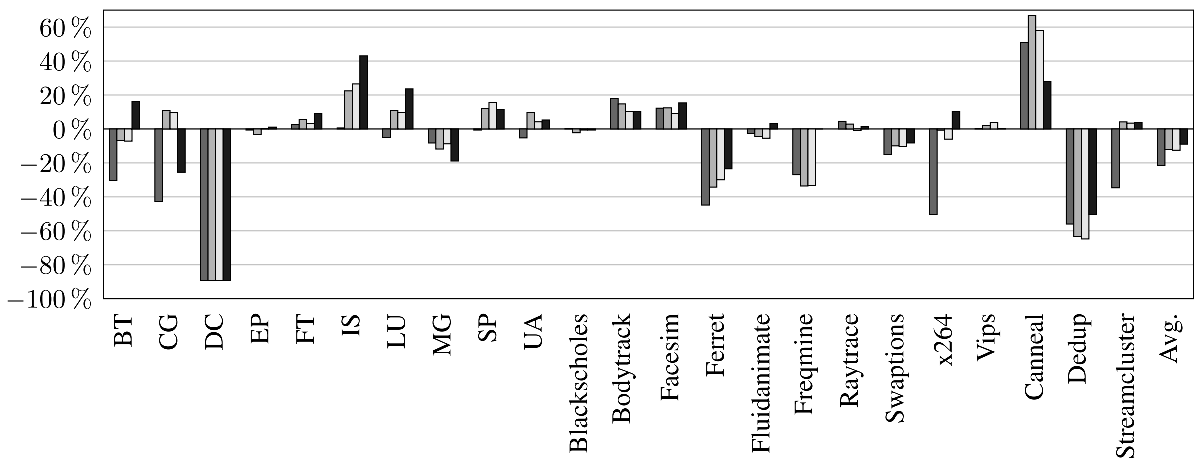
It is interesting to note that the application that had the highest reductions of both interchip traffic and L2 misses was DC, but DC was not the application with the highest execution time reduction. For instance, in LAPT, interchip traffic and L2 misses were reduced by 99.8% and



(a) Execution time in Simics/GEMS (lower is better).



(b) Interchip traffic in Simics/GEMS (lower is better).



(c) L2 cache MPKI in Simics/GEMS (lower is better).

Figure 6.25: Results in Simics/GEMS normalized to the default mapping (Equation 6.1).

89.4%, while execution time was reduced by 30.2%, 0.8% less than in its best case (MG). This clearly shows that, besides memory locality, there are other very important parameters influencing the performance of the applications, although we are changing only the mapping mechanism between different configurations. Due to this, in some occasions, the improvements do not necessarily correlate to the previously evaluated sharing patterns or exclusivity levels of the applications. One example is SP, in which, although its sharing pattern indicates that it is suitable for thread mapping (Figure 6.9), our proposals actually increase the amount of L2 cache misses due to the additional cache misses introduced by thread migrations.

In some applications, no performance improvements are expected by either thread or data mapping. Swaptions is an example of such an application. Although its sharing pattern is similar to the one of CG, its memory usage is much smaller, as almost all its data fits in the caches. Therefore, although IPM decreases interchip traffic by 10.0% in Swaptions, the absolute reduction is very small (about 9.5% of CG's) and not affected by data mapping. The behavior of Swaptions in the other mechanisms is similar.

6.4.2 Performance Results in Itanium

The execution times in Itanium can be found in Figure 6.26. As previously explained, we only implemented IPM in Itanium. CG was the application with the best reduction in execution time (39.0%). Since the memory hierarchies are very different between platforms, some applications present different results, such as DC and Ferret. The lack of hardware counters to monitor performance in Itanium makes it difficult to better understand some results, but one of the main reasons for differences comes from the absence of shared caches in Itanium. Nevertheless, most results are very similar to the ones from Simics, where NAS applications such as CG, MG, SP and UA presented the best improvements.

The results generated in Itanium show that IPM reduced execution time by an average of 13.7%, while the oracle mapping reduced execution time by 14.2% on average. The experiments with the Ondes3D application showed an execution time reduction of 30.8% using IPM, while the oracle mapping reduced execution time by 30.5%. It is important to emphasize that IPM was developed as a hardware extension and that this machine is emulating IPM's behavior with its software-managed TLB. This emulation ends up stalling the execution of the thread that generated the TLB miss, as IPM is executed in its core. This would not be an issue with the hardware implementation of IPM, as the MMU would operate in parallel to the application.

We compare IPM on Itanium to several previously mentioned techniques: Interleave, NUMA Balancing (CORBET, 2012b), the kMAF affinity framework (DIENER et al., 2014), and to mapping using TLB misses as source of information. The results of kMAF are lower than IPM due to its sampling mechanism, as kMAF needs more time to detect the memory access behavior, losing opportunities for improvements. NUMA Balancing performed a little worse than kMAF mainly because of unnecessary page migrations, since NUMA Balancing

keeps no history regarding page migrations, in contrast to kMAF. The overhead of adding more page faults in kMAF and NUMA Balancing to generate a better profile is high compared to the usage of TLB time in IPM.

The usage of TLB misses alone with no history, similarly to NUMA Balancing, resulted in too many page migrations, harming performance in several applications. The amount of page migrations using this policy is much higher than in NUMA Balancing, since the amount of TLB misses is also much higher than the amount of page faults introduced by NUMA Balancing. On the other hand, using TLB misses as a source of information to the same procedures performed by IPM results in a very good performance, since it reduces the amount of unnecessary page migrations. Nevertheless, the usage of TLB time provided a clear advantage over TLB misses in some applications, such as BT, FT, SP and Facesim. As the implementation complexity of TLB time is almost the same as TLB misses, we believe the usage of TLB time is recommended.

The comparison to the related work shows that mechanisms that perform both thread and data mapping are able to achieve better improvements than mechanisms that perform these mapping separately. It also shows that simple policies, such as the first-touch data mapping mapping, do not guarantee a good performance for all applications since they depend too much

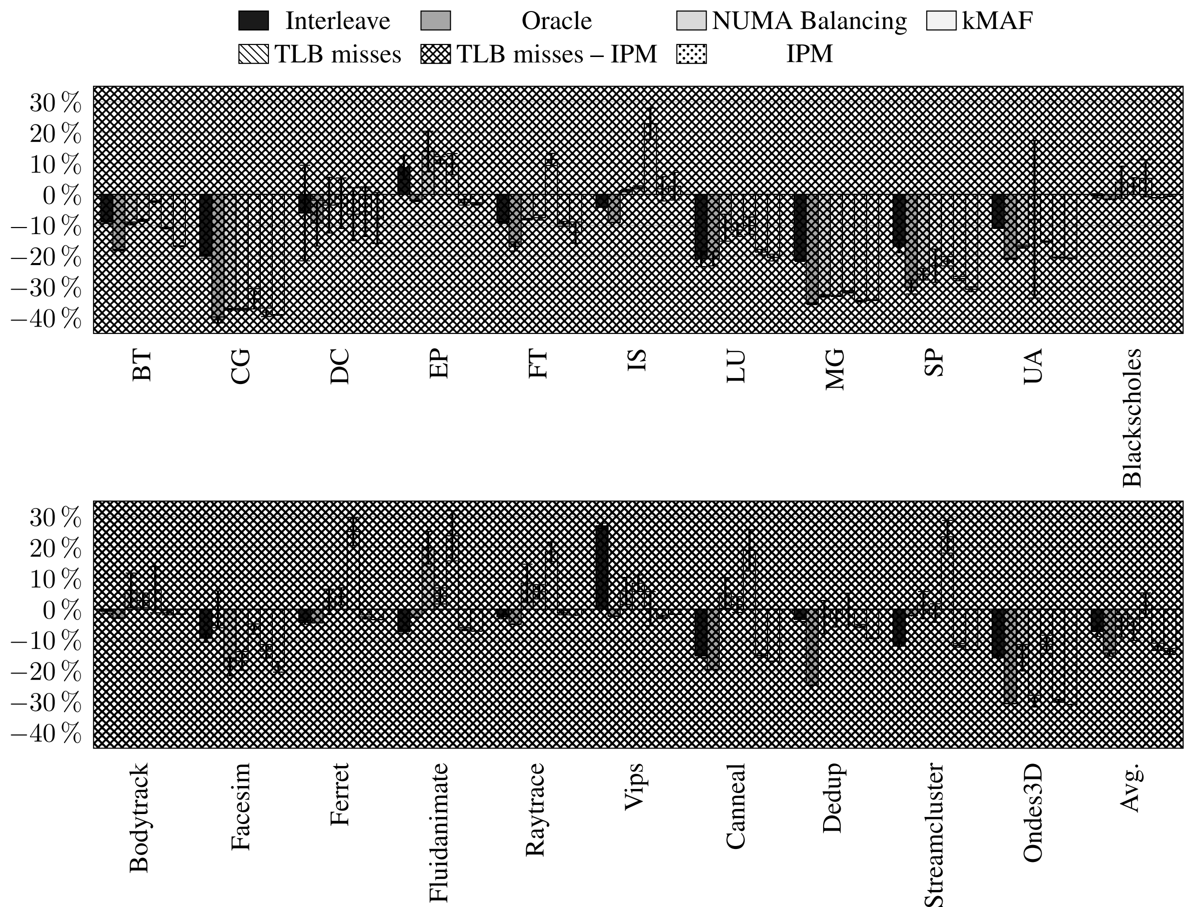


Figure 6.26: Execution time in Itanium normalized to the operating system mapping (Equation 6.1). Lower results are better.

on the characteristics of the application and how the programmer initialized the data. The interleaved mapping is also a simple policy, and provided a high variance in the results. Furthermore, the results shows that IPM, with its TLB time based metric, is able to achieve a higher accuracy and better performance than other mechanisms.

Finally, the results obtained in Itanium demonstrate that IPM works not only in a simulator, but also in a real machine. Furthermore, these results indicate that IPM is able to improve performance not only for traditional benchmarks, but also for real applications (Ondes3D). Also, as explained in Section 6.1.6, the oracle mapping was generated by inserting mapping routines directly in the source code, which shows that IPM achieves improvements as good as manually optimized code. These statements can also be extended to LAPT and SAMMU, since they operate very similarly to IPM.

6.4.3 Performance Results in Xeon32 and Xeon64

The performance results in Xeon32 and Xeon64 can be found in Figures 6.27 and 6.28, respectively. For both machines, we show the execution time, amount of interchip interconnection traffic and L2 and L3 caches MPKI. In Xeon32, CG was the application with the highest improvements, reducing execution time by 16.5%, 16.9% and 17.0% in LAPT, SAMMU and IPM. CG, as explained before, has a sharing pattern in which thread mapping is not able to improve performance, which is the reason why we only reduce interchip interconnection traffic (cache misses were actually increased). In Xeon64, SP was the application with the highest improvements, with a reduction of execution time of up to 35.9% in IPM. Due to its sharing pattern, both L3 cache misses and interchip traffic were reduced, by 61.2% and 64.1%, respectively.

We can observe how thread mapping affects data mapping by analyzing MG. The sharing pattern of MG is similar to the one of SP, thereby thread mapping affects its performance. However, the memory usage of MG is much higher than SP, such that its amount of data shared by threads is much higher than the L3 cache size, resulting in no reduction in L3 cache misses. Thread mapping improves data mapping in cases where pages are shared by multiple threads. In this way, the more appropriate thread mapping puts threads that share pages on the same NUMA node, thus reducing interchip traffic. Therefore, we are able to observe MG's potential for thread mapping by looking at interchip traffic, and not at cache misses.

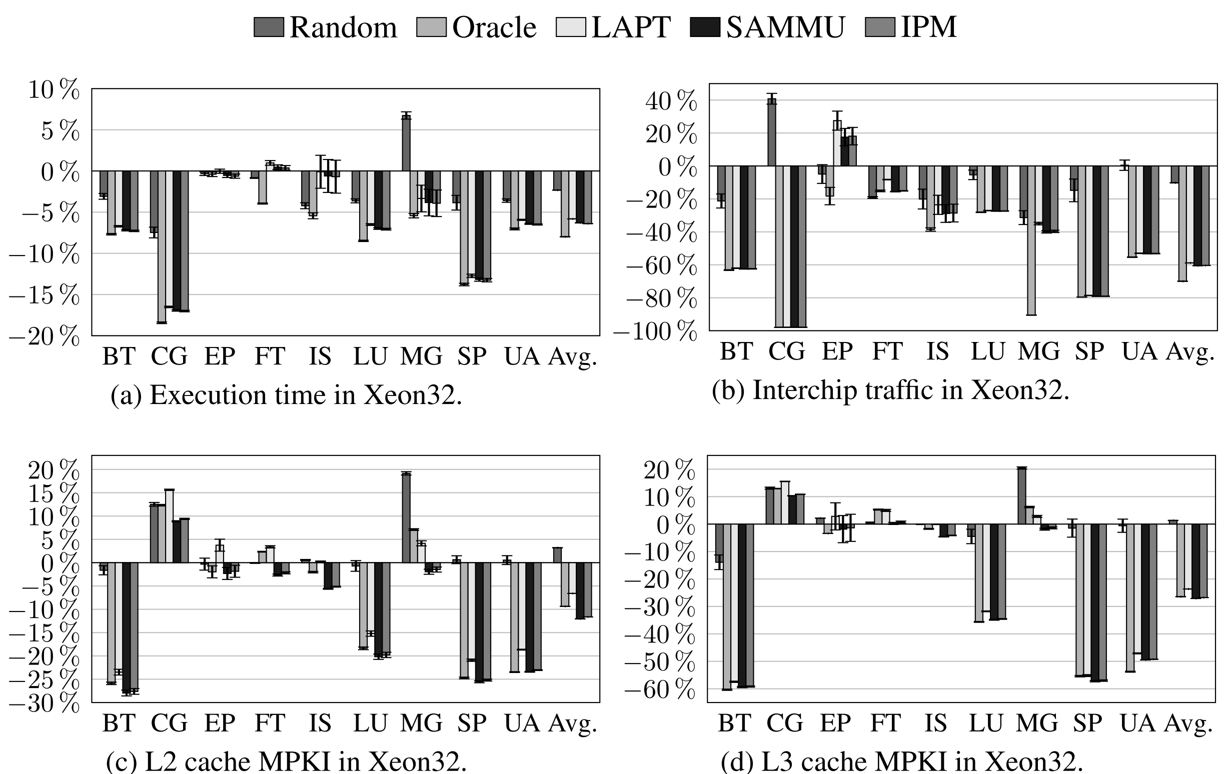
Comparing the different metrics analyzed, we observe that the highest reduction occurred in the interchip interconnection traffic, with an average reduction of 37.2% to 40.5% in Xeon64, and 58.9% to 60.5% in Xeon32. Meanwhile, execution times were reduced by an average of 12.8% to 13.3% in Xeon64, and 5.8% to 6.4% in Xeon32. The effects in total execution time are smaller because they are influenced by several factors, as opposed to interchip traffic and cache misses, which are directly influenced by better mappings.

6.4.4 Summary of Performance Results

The results obtained in Simics, Itanium and Xeon64 are better than the results obtained in Xeon32. As Simics and Xeon64 have 4 NUMA nodes (while Xeon32 has 2), the probability of finding the correct node to a page without any knowledge of the memory access pattern is only 25% on them (while it is 50% on Xeon32). Regarding Itanium, although it also has only 2 NUMA nodes, a better mapping has a bigger impact because the latency of a remote memory access is much higher than in Xeon32.

Most applications are more sensitive to data mapping than thread mapping, which can be observed in the results by the fact that the interchip traffic presented a higher reduction than cache misses. This happens because, even if an application does not share much data among its threads, each thread will still need to access its own private data, which can only be improved by data mapping. It is important to note that this does not mean that data mapping is more important than thread mapping, because the effectiveness of data mapping depends on thread mapping for shared pages.

Our proposals presented results similar to the oracle mapping, demonstrating their effectiveness. It also performed significantly better than the random mapping for most cases. This shows that the gains over the operating system are not due to the unnecessary migrations introduced by the operating system, but due to a more efficient usage of resources. Compared to other



(a) Execution time in Xeon32.
(b) Interchip traffic in Xeon32.
(c) L2 cache MPKI in Xeon32.
(d) L3 cache MPKI in Xeon32.

Figure 6.27: Performance results in Xeon32 normalized to the operating system mapping (Equation 6.1). Lower results are better.

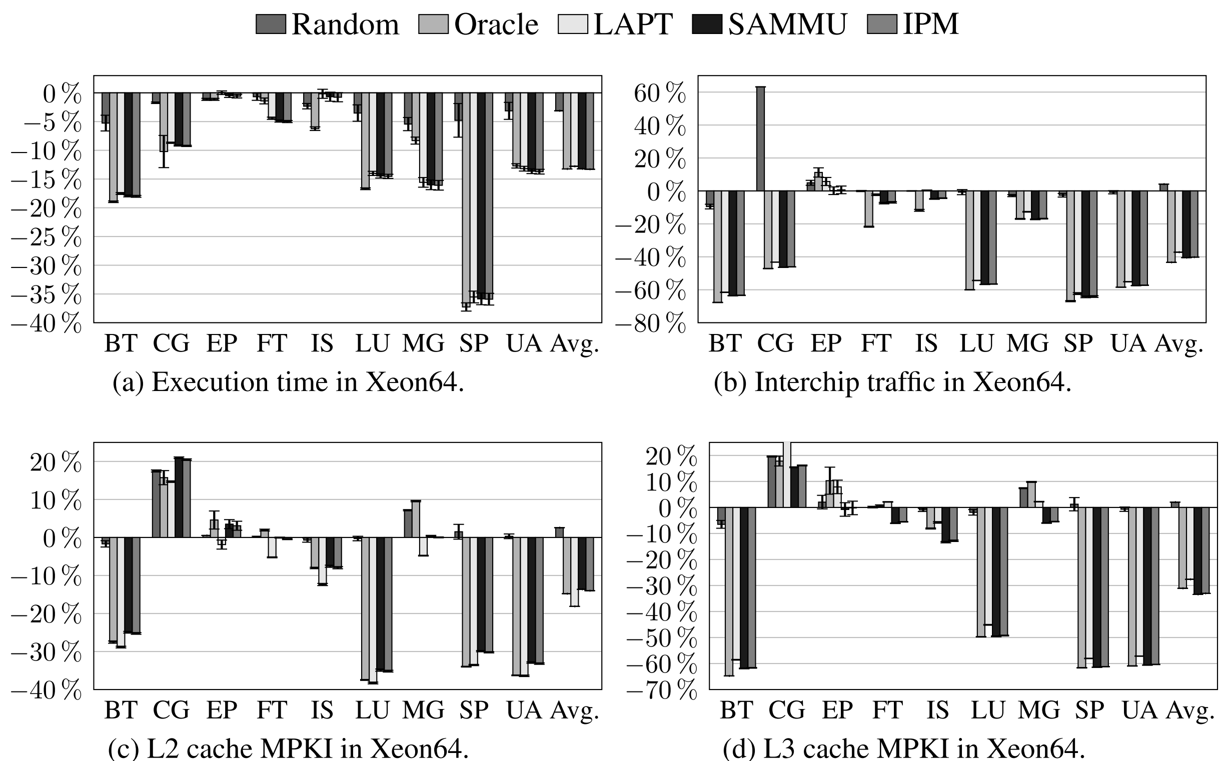
state-of-art mapping policies, our proposals presented better results for most applications.

6.5 Energy Consumption Experiments

Results of the amount of energy consumption in Xeon32 are shown in Figure 6.29. The total amount of energy consumption was reduced by an average of 5.6% to 6.1%, and up to 14.3% in SP with IPM. Energy per instruction was improved an average of 3.7% to 4.4%, and up to 12.2% in SP with IPM. This shows that our mechanisms not only saves energy by reducing the executing time, but also by providing a more efficient execution, which is an important goal for future Exascale architectures (TORRELLAS, 2009). We also measured the DIMM and processor energy separately. These measurements show a higher reduction of DIMM energy than processor energy, of 9.6% and 5.2% on average, respectively, in IPM. This happens because a sharing-aware mapping has more influence in the memory than in the processor.

6.6 Overhead

Our proposals cause an overhead on the execution of the application on the hardware and software levels. The overhead in the hardware level happens to the additional memory accesses to update mapping information. In the software level, the overhead happens due to the calcu-



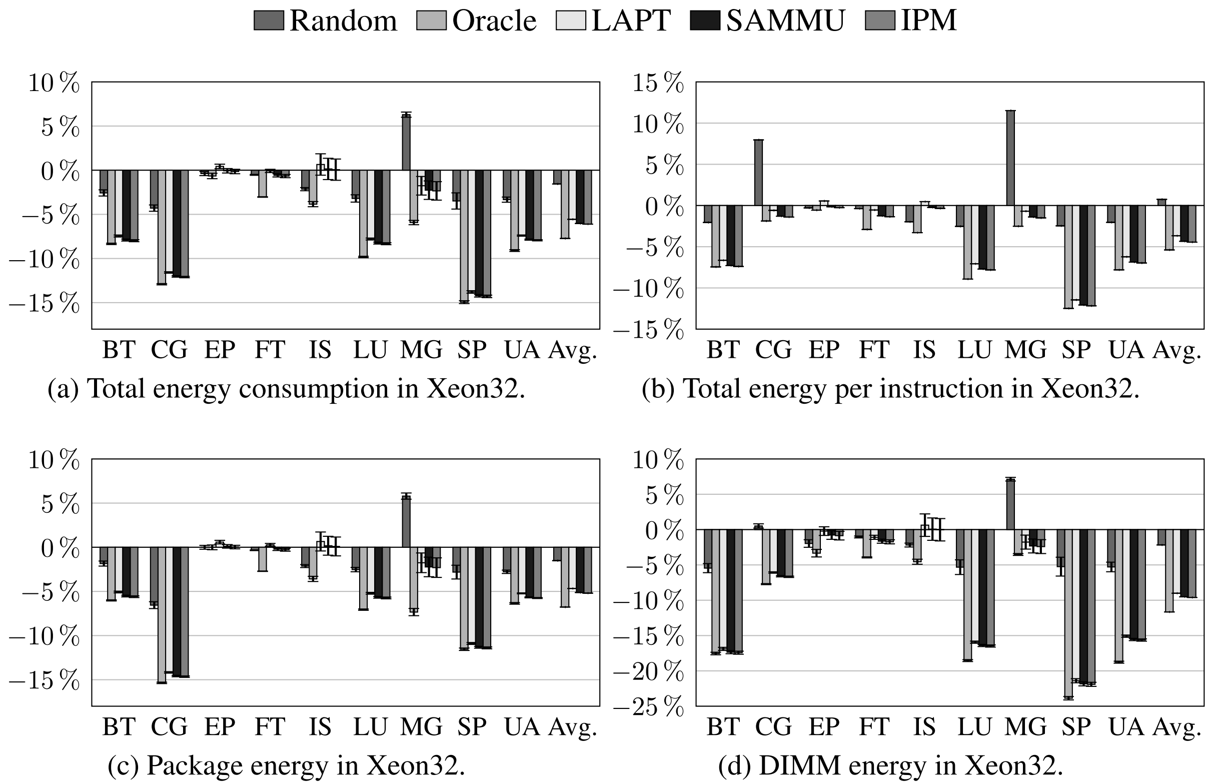
(a) Execution time in Xeon64.

(b) Interchip traffic in Xeon64.

(c) L2 cache MPKI in Xeon64.

(d) L3 cache MPKI in Xeon64.

Figure 6.28: Performance results in Xeon64 normalized to the operating system mapping (Equation 6.1). Lower results are better.



(a) Total energy consumption in Xeon32.

(b) Total energy per instruction in Xeon32.

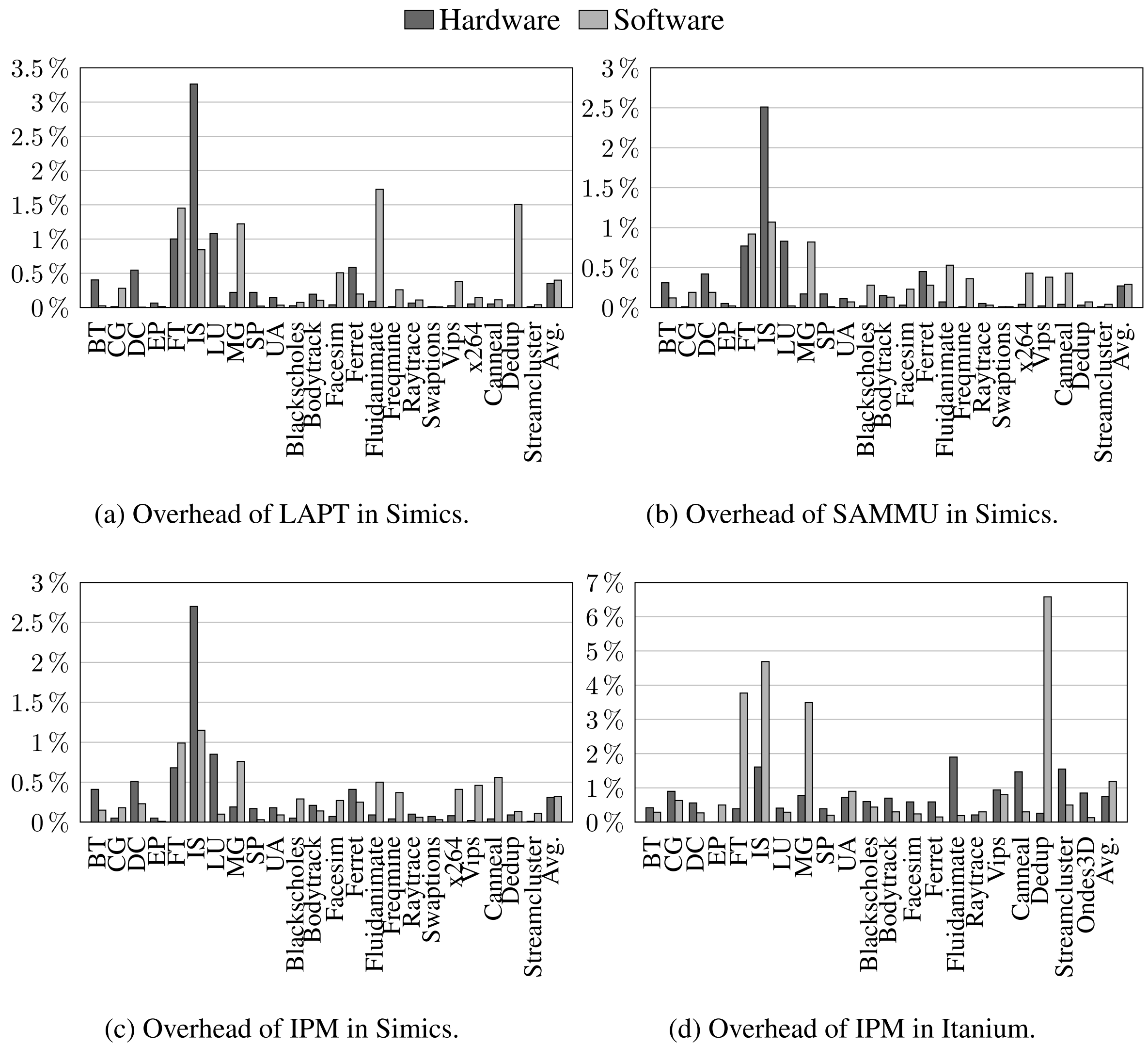
(c) Package energy in Xeon32.

(d) DIMM energy in Xeon32.

Figure 6.29: Energy consumption in Xeon32 normalized to the operating system mapping (Equation 6.1). Lower results are better.

lation of the mapping and the migrations. We evaluate the hardware overhead by running our proposals without performing any migration, and compare the execution time to the baseline without our proposals. For the software overhead, we measure the time spent to calculate the mapping and perform migrations. We only show the performance overhead in Simics and Itanium because the Xeon machines do not implement our proposals natively. Overhead is shown in Figure 6.30.

In Simics, the average performance overhead caused by the hardware was 0.36%, 0.27% and 0.31% for LAPT, SAMMU and IPM, respectively. This overhead is due to the introduction of memory accesses, such that our proposals introduced, on average, up to 1.43% additional memory transactions, in SAMMU. Regarding the amount of cycles required to handle each event (TLB miss, eviction or AC saturation in case of SAMMU), LAPT, SAMMU and IPM required 101, 109 and 255 cycles on average in Simics. IPM is the mechanism that takes longer to handle each event because it is the one that performs more memory accesses per event. This does not translate in a higher overhead because, as explained in the descriptions of the mechanisms, we implemented them to support handling only one event per core to keep a simple hardware, such that if a new event happens while another event is being handled, this new event is dropped. In the same mechanism order, they were able to handle 89.7%, 85.5% and 67.6% of all events. As we can see, IPM handles less events because it takes more time to handle each one. We also measured these statistics in Itanium running IPM, where the average



(a) Overhead of LAPT in Simics.

(b) Overhead of SAMMU in Simics.

(c) Overhead of IPM in Simics.

(d) Overhead of IPM in Itanium.

Figure 6.30: Performance overhead on the hardware and software levels. Lower results are better.

overhead of the hardware part was 0.75%, and each event required 102 cycles on average.

In Simics, application execution is not stalled while our proposals handle an event. On the other hand, in Itanium, since IPM's behavior was implemented in software in the data TLB miss handler of the kernel, application execution is stalled. The overhead in the software level was 0.40%, 0.29% and 0.32% for LAPT, SAMMU and IPM, respectively, on average in Simics. The average software overhead in Itanium was 1.19% for IPM. Some applications have a high software overhead due to the high number of migrations, which could easily be controlled by limiting the number of migrations. These results show that our proposals present only a minor overhead.

6.7 Design Space Exploration

We analyzed how our proposals behave on environments with configurations different from the previous experiments. For SAMMU, we varied the cache memory size, memory page size and interchip interconnection latency. We only present the results varying these environment parameters for SAMMU because in the other mechanisms the tendency of the results was the same. We did not vary the internal parameters of SAMMU, such as the number of bits used to encode each data structure, because they are constrained by hardware limitations. Regarding LAPT, we varied multiplication threshold used to determine if a page will be migrated, used in Equation 3.6 in Section 3.4. The analysis of SAMMU and LAPT were performed in Simics/GEMS.

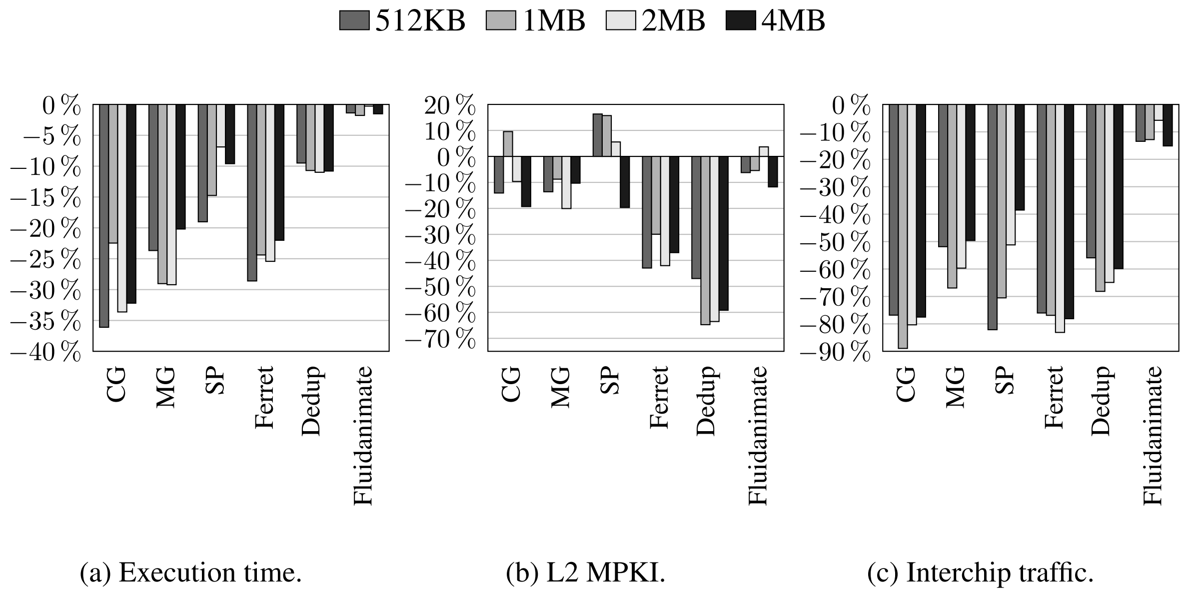
In IPM, the size of each element of the NUMA Vector, CR_{shift} and CR_{aging} depend on each other. We decided to fix the size of each NUMA Vector element to 2 bytes because it has a large range, and at the same time does not impose a high memory usage in the Page History Table. Regarding CR_{shift}, a low value would make the elapsed time (Equation 5.1 in Section 5.4.1) too large, such that it would not fit in the NUMA Vector. On the other hand, a high value for CR_{shift} would make the elapsed time lose too much precision. Based on that, we empirically determined that a good value for CR_{shift} is 13. Similarly to the parameters of SAMMU, these fixed values have several constraints. After fixing the size of the NUMA Vector and CR_{shift}, we analyzed how the aging (CR_{aging}) and the migration threshold (CR_{mig}) affect performance. The analysis of IPM were performed in Itanium.

In the next sections, we show results varying these parameters. Only a subset of the applications were used in these experiments, since the tendency of the results is the same for most applications.

6.7.1 Varying the Cache Memory Size (SAMMU)

The cache memory size influences the performance improvements obtained with sharing-aware mapping because it affects the amount of cached shared data and the accesses to main memory. The previous experiments were performed with L2 cache memories with 1 MByte of capacity and a 5 cycles latency. We now show results for caches with 512 KBytes, 2 MBytes, and 4 MBytes, and latencies of 4, 6, and 6 cycles, respectively, as calculated using CACTI (THOZIYOOR et al., 2008).

The results of varying the cache size are shown in Fig. 6.31. The results follow the same pattern of the previous experiments: the reduction of execution time depends on the reduction of interchip interconnection traffic. We can observe that SAMMU was able to improve performance for all applications and all cache sizes. Our other proposals had similar results. This shows that our proposals can handle architectures regardless of their cache sizes.



(a) Execution time. (b) L2 MPKI. (c) Interchip traffic.

Figure 6.31: Varying L2 cache memory sizes in Simics/GEMS with SAMMU. Results are normalized to the default mapping with the corresponding cache size. Lower results are better.

6.7.2 Varying the Memory Page Size (SAMMU)

Another important parameter for SAMMU is the page size, since it influences the number of TLB misses and, thereby, evictions. In this context, the normalized execution time and interchip interconnection traffic measured with different page sizes are shown in Figure 6.32a and 6.32b (the previous experiments in Simics were performed with a 4 KBytes page size). We also show the exclusivity level (more details in Section 2.1.3.2) calculated in the simulations, in Figure 6.32d. The higher the exclusivity level of an application, the higher its potential for sharing-aware data mapping. The TLB miss rate is also shown, in Figure 6.32c.

TLB miss and eviction rates drop considerably when the page size increases, decreasing the number of updates to the Sharing Matrix and Page History Table. Since SAMMU detects the sharing pattern in the page level granularity, increasing page size can also lead to the detection of different patterns. The analysis of the exclusivity level shows that, for all applications except Ferret, the interference of accesses in different offsets is low when the page size is 1 KByte or 4 KBytes, but increases noticeably in larger page sizes. In relation to data mapping, the lower exclusivity level with larger page sizes tends to limit performance improvements, since there are more threads executing in cores from different NUMA nodes when the page size is larger, which is reflected in the lower reduction of interchip traffic and, thereby, in performance. In some cases, the best performance improvements happened with an intermediate page size, such as for MG. This happened because pages were migrated earlier during the execution, such that the benefits of the improved data mapping were taking effect earlier too.

It is important to note that the memory footprint of the applications used in the simulations is very small due to the small input size required by simulation time constraints. Applications with

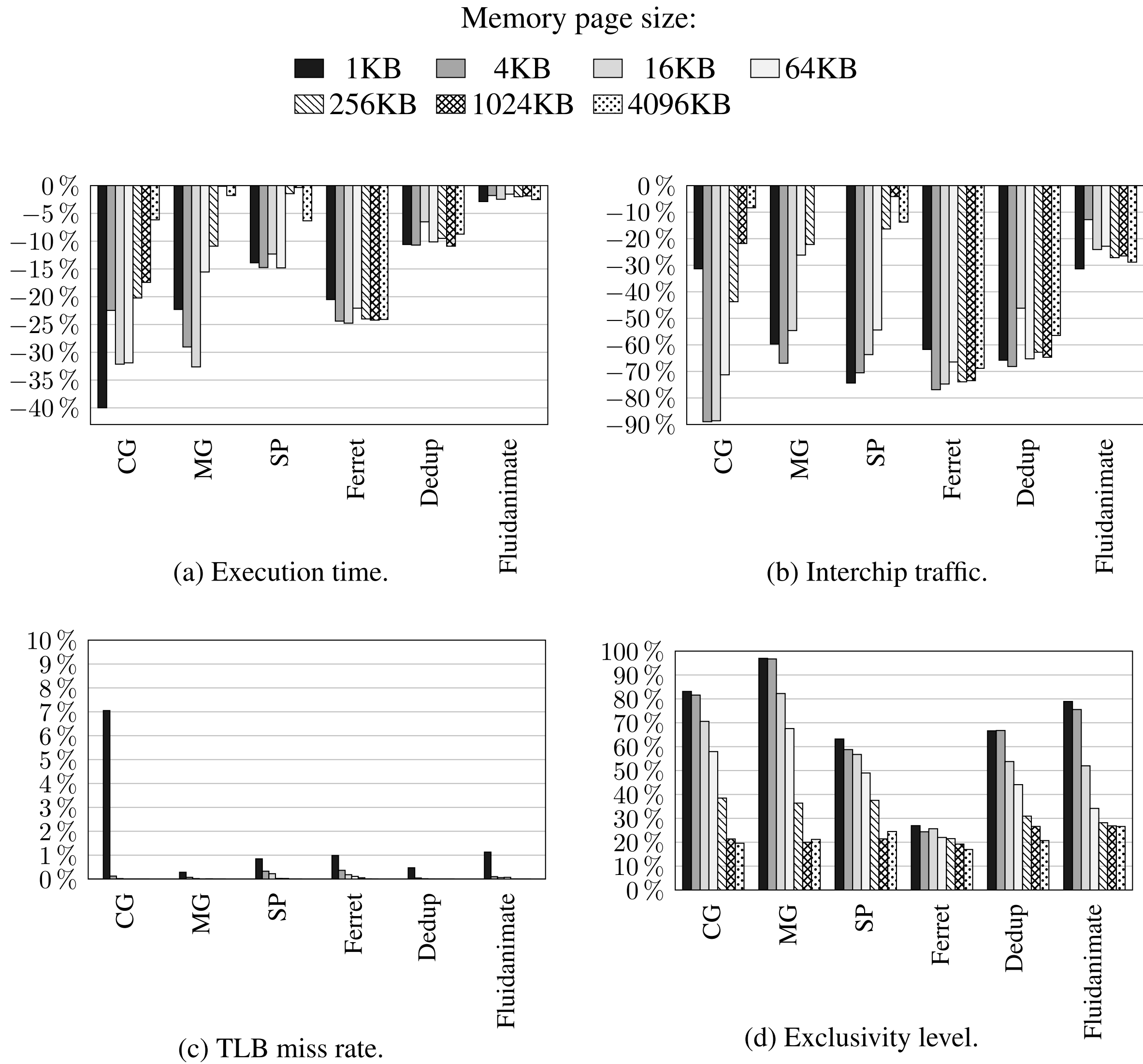
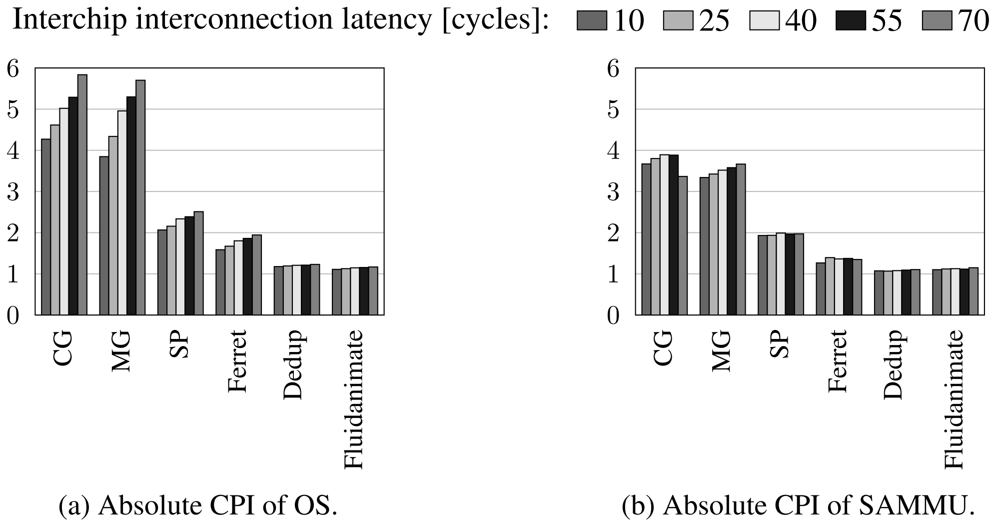


Figure 6.32: Varying memory page sizes in Simics/GEMS with SAMMU. Results of Figures 6.32a and 6.32b are normalized to the default mapping with the corresponding page size. In Figures 6.32a and 6.32b, lower results are better.

higher memory footprints keep similar exclusivity levels with larger page sizes (DIENER et al., 2014). Therefore, SAMMU would maintain similar performance improvements with larger page sizes for applications that use large amounts of memory. Our other proposals presented results similar to SAMMU, showing that they can handle a wide range of page sizes.

6.7.3 Varying the Interchip Interconnection Latency (SAMMU)

The interchip interconnection latency influences the time it takes to send cache coherence messages, such as cache line invalidations, as well as cache line transfers between caches, affecting thread mapping. Regarding data mapping, the interchip interconnection latency influences the time it takes to perform remote memory accesses. Local memory accesses are not



(a) Absolute CPI of OS.

(b) Absolute CPI of SAMMU.

Figure 6.33: Varying interchip interconnection latency in Simics/GEMS with SAMMU. Lower results are better.

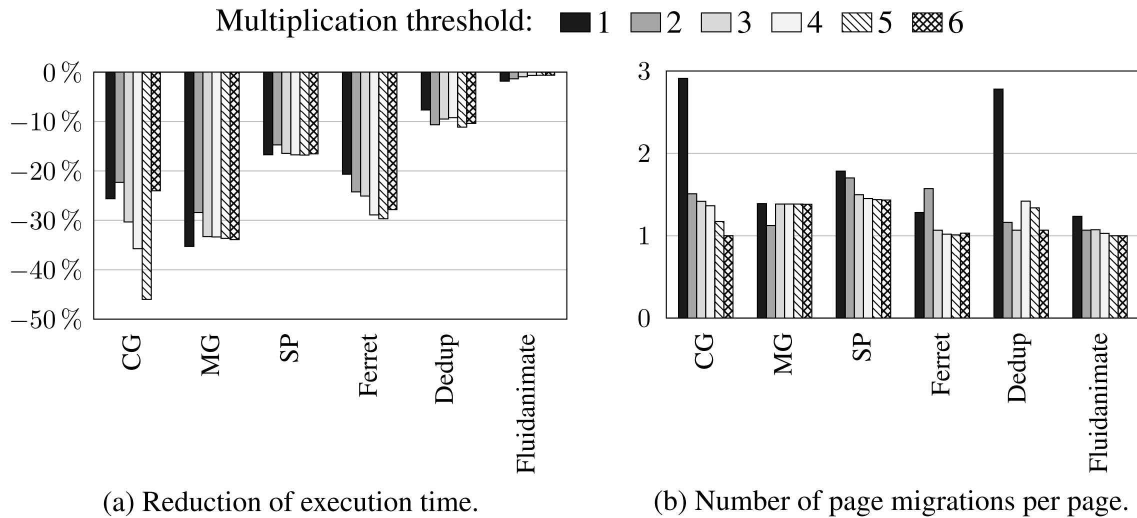
affected. In this section, we briefly evaluate how SAMMU behaves with latencies between 10 and 70 cycles (previous experiments used a latency of 40 cycles).

The results obtained with the interchip interconnection latency variation are shown in Figure 6.33. In the absolute values, we chose to show cycles per instruction (CPI) instead of execution time due to the high difference of execution time between the applications. We can observe that the CPI measured using the operating system increases significantly with the increase of the interchip latency for most applications. On the other hand, the CPI obtained by SAMMU suffers a much lower impact from latency increase. SAMMU is less influenced by this latency because it is able to reduce the number of coherence messages, cache-to-cache transfers and remote memory accesses. Our other proposals presented similar results.

6.7.4 Varying the Multiplication Threshold (LAPT)

In the previous experiments, we used Equation 3.6 to determine if a page should migrate to another node. The equation determines that a page migration will happen only if the value of the counter of the node with highest NV_P is greater or equal to twice the average value. In this section, we analyze the sensitivity of the mechanism to multiplication thresholds other than 2. We vary the threshold from 1 to 6 and evaluate the impact on execution time and number of page migrations. The execution time is shown in Figure 6.34a, and the number of migrations per page is shown in Figure 6.34b.

We can observe the tendency that, when increasing the multiplication factor, the number of page migrations decreases. This is the expected behavior, since the higher the multiplication factor in Equation 3.6, more accesses from a node are required to trigger a page migration. The applications most affected by the number of page migrations are CG and Ferret, where LAPT



(a) Reduction of execution time.

(b) Number of page migrations per page.

Figure 6.34: Varying the multiplication threshold. Figure 6.34a is normalized to the default mapping, where lower results are better.

performs better with less page migrations. This happened not only due to a lower overhead of copying the pages across nodes, but also due to lowering its side effects, such as the cache misses and pollution (the address of the page changes when moving to another node) and interconnection traffic of transferring the page. In CG, the best improvements happen when the threshold is set to 5. When the threshold is set to 6, the performance decreases because of two reasons: (i) some important page migrations did not happen; (ii) pages take more time to migrate. In the other applications, the influence of the multiplication factor is very low.

6.7.5 Analyzing the Impact of Aging (CR_{aging} in IPM)

The execution time varying the aging value is shown in Figure 6.35. The results are normalized to the results of CR_{aging} set to 7. The higher the value of CR_{aging}, less aging is

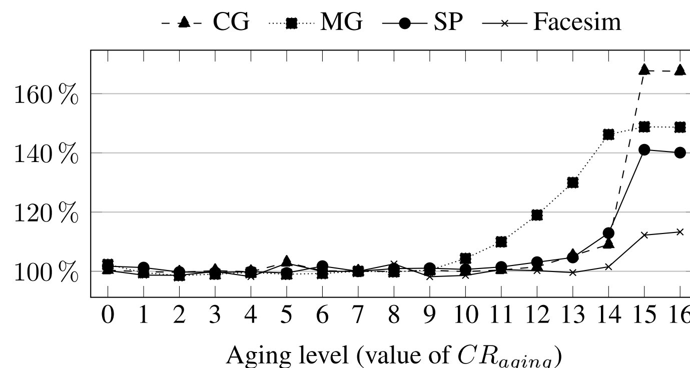


Figure 6.35: Normalized execution time varying the aging in IPM running in Itanium. The higher CR_{aging}, less aging. Lower results are better.

performed (Eq. 5.6 in Section 5.4.3). In our experiments, we observed that an aggressive aging (low CR_{aging}) did not have a negative impact on performance, because the migrations did not increase significantly, as, in these applications, most memory pages are accessed by a single NUMA node (DIENER et al., 2014). On the other hand, we observed a negative impact on performance when using a conservative aging (high CR_{aging}) because it increased too much the time to start migrating data, since the initial values of the NUMA Vector are set according to the aging (Equation 5.9 in Section 5.4.3).

6.7.6 Analyzing the Impact of the Migration Threshold (CR_{mig} in IPM)

The execution time varying the migration threshold value is shown in Figure 6.36. The results are normalized to the results of CR_{mig} set to 2. The higher the value of CR_{mig}, the more difficult it is to migrate a page (Equation 5.8 in Section 5.4.3). We observed that an aggressive threshold (low CR_{mig}) did not lead to too many migrations, not harming performance. The reason is the same as in the evaluation of the aging: in these applications, most memory pages are accessed by a single NUMA node (DIENER et al., 2014). However, conservative values of the threshold (high CR_{mig}) made it too difficult to migrate the pages, losing opportunities to improve performance.

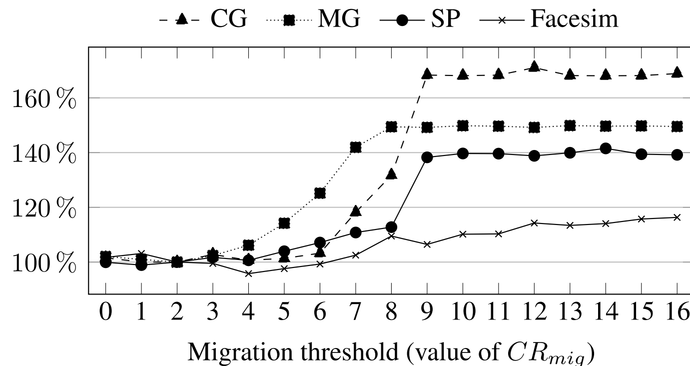


Figure 6.36: Normalized execution time varying the migration threshold in IPM running in Itanium. The higher CR_{mig}, less migrations. Lower results are better.

7 CONCLUSIONS AND FUTURE WORK

We begin this chapter with the conclusions of this thesis, followed by a discussion of future work. Finally, we show the publications referent to this thesis.

7.1 Conclusions

The memory locality is one of the most important aspects to be considered when designing a new architecture. In the past, when single core architectures were common, memory locality was increased by adding a simple cache memory hierarchy. With the introduction of multicore architectures, the memory hierarchy had to evolve to able to provide the necessary bandwidth to several cores operating in parallel. With this evolution, memory hierarchies started to present several caches in the same level, some levels shared by multiple cores, and other private to a core. Another important step was the incorporation of a memory controller inside the processor, in which multiprocessor systems presented NUMA characteristics. Due to the introduction of such technologies, the performance of memory hierarchies and the systems as a whole were even more dependent on memory locality. In this context, techniques such as sharing-aware thread and data mapping are able to increase memory locality.

Related work on the area of sharing-aware mapping have been proposed, with a wide variety of characteristics and features. The majority of the proposals perform only static mapping, which are able to handle only applications whose memory access behavior keeps the same along different executions. Most work also only handle thread or data mapping alone, not both together. The related work that were able to handle both thread and data mappings and operate online, during the execution of the application, had a high trade-off between accuracy and overhead. To achieve a higher accuracy, they have to increase the overhead of their memory access behavior detection as well. With this analysis, we identified a gap in the state-of-art: there was no online mechanism that could achieve a high accuracy.

In this thesis, we proposed novel solutions to fulfill this gap of the related work. Our solutions make use of the MMU present in current processors, as the MMU already has access to information regarding all memory accesses performed by each core. The simplest mechanism we proposed is LAPT, which tracks the threads that access each page and estimates the amount of pages shared by the threads. We also proposed SAMMU, which, besides what LAPT does, also estimates the amount of accesses performed to each page from all NUMA nodes. Finally, we proposed IPM, which provides information similar to SAMMU, but has a simpler procedure and uses information regarding the time each page has its entry cached in the TLB instead of the amount of memory accesses. With the proposed hardware support, our mechanisms are able to offer a high accuracy with a low performance overhead.

We have evaluated our mechanisms in three different environments: a full system simulator, a real machine with software-managed TLB and a trace driven evaluation in two real machines

with hardware-managed TLBs. Our mechanisms provided substantial performed improvements in all environments. In the full system simulator, LAPT, SAMMU and IPM reduced execution timer by an average of 10.1%, 10.9% and 11.4%, respectively. In the same mechanism order, execution time was reduced by 5.8%, 6.3% and 6.4% in a real machine with two NUMA nodes and hardware-managed TLB, and 12.8%, 13.2% and 13.3% in a real machine with 4 NUMA nodes. As expected, we got better results with more NUMA nodes, since the memory hierarchy is more complex. In the real machine with software-managed TLB, execution time was reduced by 13.7% on average. Although this machine has only two NUMA nodes, its internode interconnection is very slow, such that a better mapping is translated to more impressive gains.

Besides the execution time, we analyzed the reasons why our proposals achieved such execution time reduction. The two aspects that were measured for this analysis were cache misses and interchip interconnection traffic. In the real machine with 2 NUMA nodes, L3 cache misses were reduced by an average of 23.6%, 27.1% and 26.8% for LAPT, SAMMU and IPM, respectively. In the real machine with 4 NUMA nodes, L3 cache misses were reduced by 27.6%, 33.4% and 33.0% on average. Cache misses were reduced due to the better usage of shared cache memories, storing data shared by several cores in the same cache. In the real machine with 2 NUMA nodes, interchip interconnection traffic was reduced by an average of 58.9%, 60.5% and 60.4% for LAPT, SAMMU and IPM, respectively. In the real machine with 4 NUMA nodes, interchip interconnection traffic was reduced by 37.2%, 40.5% and 40.2% on average. Interchip traffic was reduced due to less cache coherence transactions and less remote memory accesses. Another important aspect that was improved was the energy consumption efficiency, specially in the current context of pursuing exascale computing. Energy consumption efficiency was improved by 3.7%, 4.3% and 4.4% on average. We compared our mechanisms to related work, where we proved that our mechanisms, which achieve a higher accuracy with low overhead, are able to provide better improvements.

The improvements were tightly dependent on the characteristics of the parallel applications and architectures. In applications whose threads have a very uniform data sharing among them, or whose pages have similar amount of accesses from all NUMA nodes (low exclusivity level), our proposals provided low or no improvements. This is expected, as no mapping in such applications is able to optimize any resource usage. On the other hand, in applications whose threads share lots of data within a subgroup of threads, or whose pages have most of accesses from one NUMA node, our proposals presented high improvements. In such applications, our proposals reduced execution by up to 39.0% (IPM running the CG application in the real machine that has a software-managed TLB).

7.2 Future Work

As future work, we intend to evaluate and propose solutions to improve memory locality targeting more recent systems, such as the Xeon Phi, which also feature different data access

latencies between the cores. Also, we want to expand the mapping to consider other aspects of the architectures, such as the usage of functional units and load balancing, and evaluate the mapping running multiple parallel applications simultaneously. Another possibility is to consider mapping in heterogeneous computing, in order to identify the characteristics of a parallel application and map its threads to the cores that most suit their characteristics.

7.3 Publications

The following papers were published during the Ph.D.:

1. Eduardo H. M. Cruz, Matthias Diener, Marco A. Z. Alves, Laércio L. Pilla, Philippe O. A. Navaux. **LAPT: A Locality-Aware Page Table for thread and data mapping**. Accepted for publication in Parallel Computing (PARCO), 2016.
2. Emmanuell D. Carreño, Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux. **Automatic Communication Optimization of Parallel Applications in Public Clouds**. Accepted for publication in IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2016.
3. Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Philippe O. A. Navaux. **Communication in Shared Memory: Concepts, Definitions, and Efficient Detection**. Accepted for publication in Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2016.
4. Eduardo H. M. Cruz, Matthias Diener, Philippe O. A. Navaux. **Communication-Aware Thread Mapping Using the Translation Lookaside Buffer**. Concurrency and Computation: Practice and Experience, v. 22, n. 6, 2015.
5. Eduardo H. M. Cruz, Matthias Diener, Laércio L. Pilla, Philippe O. A. Navaux. **An Efficient Algorithm for Communication-Based Task Mapping**. Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2015. **(Best paper award)**
6. Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Mohammad S. Alhakeem, Philippe O. A. Navaux, Hans-Ulrich Heiss. **Locality and Balance for Communication-Aware Thread Mapping in Multicore Systems**. Euro-Par, 2015.
7. Matthias Diener, Eduardo H. M. Cruz, Laércio L. Pilla, Fabrice Dupros, Philippe O. A. Navaux. **Characterizing Communication and Page Usage of Parallel Applications for Thread and Data Mapping**. Performance Evaluation, 2015.
8. Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux, Anselm Busse, Hans-Ulrich Heiss. **Communication-Aware Process and Thread Mapping Using Online Communication Detection**. Journal of Parallel Computing (PARCO), March 2015.
9. Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux. **Locality vs. Balance: Exploring Data Mapping Policies on NUMA Systems**. Euromicro International Con-

ference on Parallel, Distributed, and Network-Based Processing (PDP), March 2015.

10. Eduardo H. M. Cruz, Matthias Diener, Marco A. Z. Alves, Philippe O. A. Navaux. **Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols**. Journal of Parallel and Distributed Computing (JPDC), v. 74, n. 3, March 2014.
11. Eduardo H. M. Cruz, Matthias Diener, Marco A. Z. Alves, Laércio L. Pilla, Philippe O. A. Navaux. **Optimizing Memory Locality Using a Locality-Aware Page Table**. International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2014.
12. Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux, Anselm Busse, Hans-Ulrich Heiss. **kMAF: Automatic Kernel-Level Management of Thread and Data Affinity**. International Conference on Parallel Architectures and Compilation Techniques (PACT), August 2014.
13. Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux. **Communication-Based Mapping using Shared Pages**. IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2013.

The following papers are currently submitted and under review:

1. Eduardo H. M. Cruz, Matthias Diener, Laércio L. Pilla, Philippe O. A. Navaux. **Hardware-Assisted Thread and Data Mapping in Hierarchical Multi-Core Architectures**. ACM Transactions on Architecture and Code Optimization (TACO).
2. Eduardo H. M. Cruz, Matthias Diener, Laércio L. Pilla, Philippe O. A. Navaux. **A Sharing-Aware Memory Management Unit for Online Mapping in Multi-Core Architectures**. Euro-Par, 2016.

APPENDIX A THREAD MAPPING ALGORITHM

Parallel architectures introduce a complex hierarchy to allow efficient memory accesses and communication among tasks. Within a machine, the hierarchy consists of several cache memory levels (private or shared), as well as Non-Uniform Memory Access (NUMA) behavior, in which memory banks are divided into NUMA nodes. In clusters and grids, the hierarchy introduces routers, switches and network links with different latencies, bandwidths and topologies.

The different levels of the hierarchy influence the communication performance between the tasks of a parallel application (WANG et al., 2012). Communication through a shared cache memory or intra-chip interconnection is faster than communication between processors due to the slower inter-chip interconnections (CRUZ et al., 2014a). Likewise, communication within a machine is faster than the communication between nodes in a cluster or grid. In this context, the mapping of tasks to processing units (PUs) plays a key role in the performance of parallel applications (ZHAI; SHENG; HE, 2011). Tasks that communicate intensely should be mapped to PUs close together in the hierarchy.

The communication-based task mapping problem can be defined as follows (CRUZ et al., 2014a). Consider two graphs, one representing the parallel application, and one representing the parallel architecture. In the application graph, vertices represent tasks and edges represent the amount of communication between them. In the architecture graph, vertices represent the machine components, including the PUs, cache memories, NUMA nodes, network routers, switches, and others organized hierarchically, while edges represent the links' bandwidth and latency. The task mapping problem consists of finding a mapping of the tasks in the application graph to the PUs in the architecture graph, such that the total communication cost is minimized.

The complexity of finding an optimal mapping is NP-Hard (BOKHARI, 1981). Due to the high number of tasks and PUs, finding an optimal mapping for an application is unfeasible. Heuristics are therefore employed to compute an approximation of the optimal mapping. However, current mapping algorithms still present a high execution time, since they were developed focusing on static mappings and are mostly based on complex analysis of the graphs. This reduces their applicability, especially for online mapping, since their overhead may harm performance.

In this appedix, we propose *EagerMap* (CRUZ et al., 2015), an efficient algorithm to generate communication-based task mappings. EagerMap is designed to work with symmetric tree hierarchies. It coarsens the application graph by grouping tasks that communicate intensely. This coarsening follows the topology of the architecture hierarchy. Based on observations of application behavior, we propose an efficient greedy strategy to generate each group. It achieves a high accuracy and is faster than other current approaches. After the coarsening, we map the group graph to the architecture graph.

A.1 Related Work

Previous studies evaluate the impact of task mapping considering the communication (WANG et al., 2012), showing that it can influence several hardware resources. In shared memory environments, communication-based task mapping reduces execution time, cache misses and interconnection traffic (CRUZ et al., 2014a). In the context of cluster and grid environments, mapping tasks that communicate to the same computing node reduces network traffic and execution time (BRANDFASS; ALRUTZ; GERHOLD, 2012; ITO; GOTO; ONO, 2013). Communication cost can also be minimized in virtualized environments (SONNEK et al., 2010), demonstrating its importance for cloud computing.

Several mapping algorithms have been proposed to optimize communication. Most traditional algorithms are based on graph partitioning, such as Zoltan (DEVINE et al., 2006) and Scotch (PELLEGRINI, 1994). They use generic graphs to represent the topology, and have a complexity of $O(N^3)$ (HOEFLER; SNIR, 2011). Tree representations are used in TreeMatch (JEANNOT; MERCIER; TESSIER, 2014), which can lead to more optimized algorithms. However, the algorithm used in TreeMatch to group tasks has an exponential complexity because it generates all possible groups of tasks for each level of the memory hierarchy. Furthermore, TreeMatch does not support mapping multiple threads to the same PU.

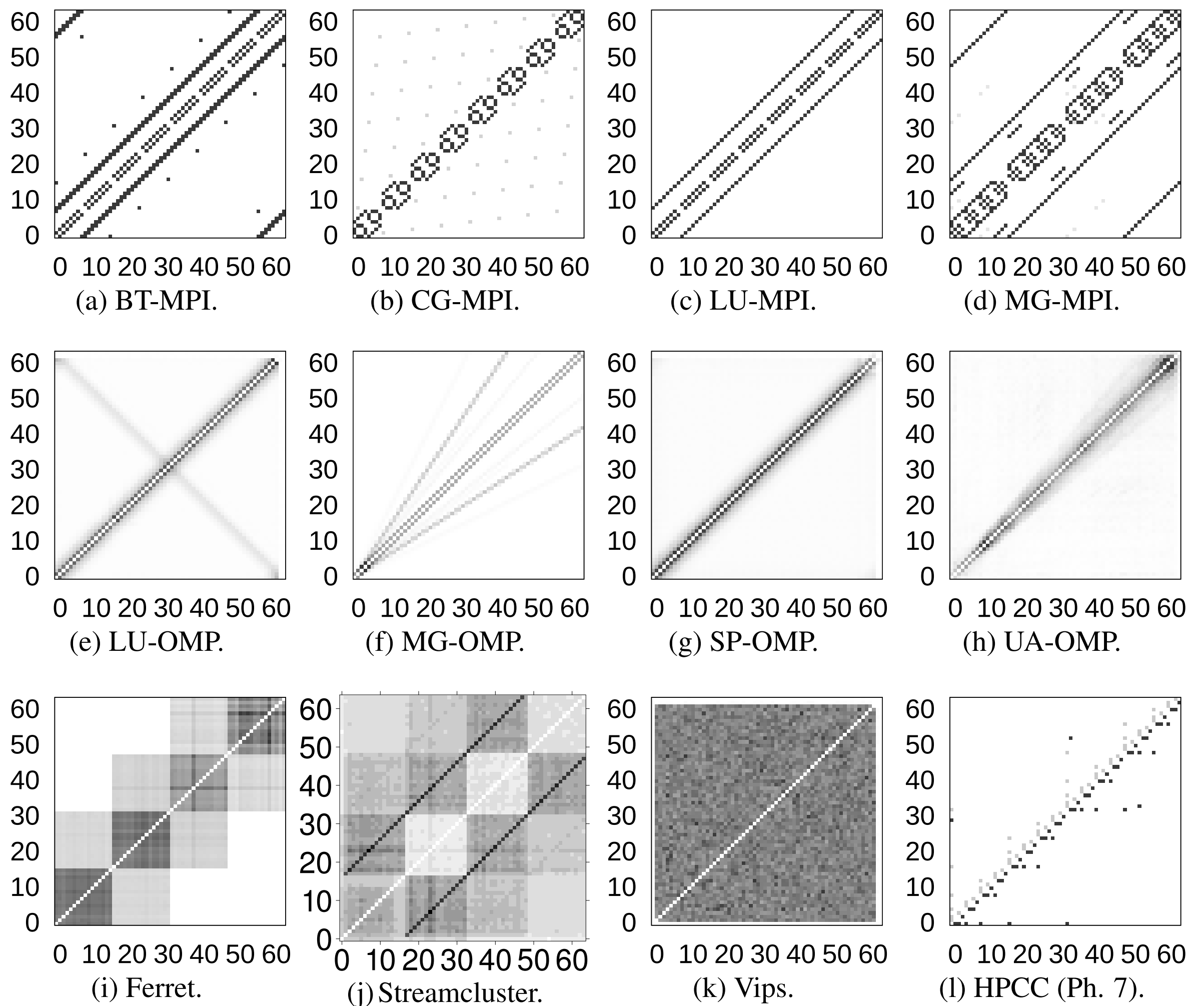
MPIPP (CHEN et al., 2006) is a framework to find optimized mappings for MPI-based applications. MPIPP initially maps each task to a random PU. At each iteration, MPIPP selects pairs of tasks to exchange PUs to reduce communication cost as much as possible. The accuracy of MPIPP depends on the initial random mapping. Our previous work (CRUZ et al., 2014a) uses Edmonds' graph matching algorithm to calculate mappings, but is limited to environments where the number of tasks and PUs is a power of two.

A.2 EagerMap – Greedy Hierarchical Mapping

EagerMap receives two pieces of input, a communication matrix containing the amount of communication between each pair of tasks as well as a description of the architecture hierarchy, and outputs which PU executes each task. To represent the architecture hierarchy, we use a tree, in which the vertices represent objects such as PUs and cache memories, and the edges represent the links between them. Our task grouping is performed with an efficient greedy strategy that is based on an analysis of the communication pattern of parallel applications. Figure A.1 depicts the different communication patterns of several parallel benchmark suites (BAILEY et al., 1991; JIN; FRUMKIN; YAN, 1999; LUSZCZEK et al., 2005; BIENIA et al., 2008), which we obtained with the methodology described in Section A.3.1.2. We observe three essential characteristics in the communication behavior of the applications that need to be considered for an efficient mapping strategy:

1. There are two types of communication behavior: structured and unstructured communication. In applications with structured communication, each task communicates more with a subgroup of tasks, such that mapping these subgroups to PUs nearby in the hierarchy can improve performance. In Figure A.1, all applications except Vips show structured communication patterns. Our mapping algorithm is designed to handle structured communication patterns, because in applications with unstructured communication, there may not be a task mapping that can improve performance.

2. In applications with structured communication patterns, the size of the subgroups with intense internal communication is usually small when compared to the total number of tasks in the parallel application. For instance, in the communication pattern of CG-MPI (Figure A.1b), subgroups of 8 tasks communicate intensely, out of 64 tasks in total.



(a) BT-MPI. (b) CG-MPI. (c) LU-MPI. (d) MG-MPI.

(e) LU-OMP. (f) MG-OMP. (g) SP-OMP. (h) UA-OMP.

(i) Ferret. (j) Streamcluster. (k) Vips. (l) HPCC (Ph. 7).

Figure A.1: Example communication matrices for parallel applications consisting of 64 tasks. Axes represent task IDs. Cells show the amount of communication between tasks. Darker cells indicate more communication.

3. The amount of communication within each subgroup is much higher than the amount of communication between different subgroups.

In this section, we describe EagerMap in detail, give an example of its operation and discuss its complexity.

A.2.1 Description of the EagerMap Algorithm

The algorithm requires two variables to be initialized previously: $nLevels$ and $execElInLevel$. $nLevels$ is the number of shared levels of the architecture hierarchy plus two. This addition is required to create a level to represent application tasks (level 0) and another level to represent the processing units (level 1). $execElInLevel$ is a vector with $nLevels$ positions. $execElInLevel[0]$ is not used. $execElInLevel[1]$ contains the number of processing units. For positions i, such that $1 < i < nLevels$, the value is the number of hardware objects on the respective architecture hierarchy level. Hardware objects are cores, caches, processors and NUMA nodes, among others. For instance, $execElInLevel[2]$ can be the number of cores, $execElInLevel[3]$ the number of last level caches, and $execElInLevel[nLevels - 1]$ the number of NUMA nodes. Since private levels of the architecture hierarchy are not important for our mapping strategy, we only consider the shared levels when preparing both $nLevels$ and $execElInLevel$.

A.2.1.1 Top Level Algorithm

The top level mapping algorithm is shown in Algorithm A.1. The algorithm calculates the mapping for each level on the architecture hierarchy. The $groups$ variable represents the groups of elements for the level being processed. The $previousGroups$ variable represents the groups of elements of the previous level. First, it initializes $groups$ with the application tasks (loop in line 2). Afterwards, it iterates over all levels on the architecture hierarchy, in line 8. After generating the groups of tasks for a level (line 10) (explained in Section A.2.1.2), the algorithm generates a new communication matrix (line 12, discussed in Section A.2.1.3). This step is necessary since we consider each group of tasks as the base element for mapping on the next hierarchy level.

The $groups$ variable implicitly generates a tree of groups. Level 0 represents the tasks. Level 1 represents groups of tasks. Level 2 represents groups of groups of tasks. In other words, on each level a new application graph is generated by coarsening the previous level. After the loop in line 8 finishes, the $groups$ variable represents the hierarchy level $nLevels - 1$ and contains $nElements$ elements. We set up $rootGroup$ to point to these elements of the highest level (for loop in line 17). Finally, the algorithm maps the tree that represents the tasks, $rootGroup$, to the tree that represents the architecture topology, $archTopologyRoot$. This procedure is explained in Section A.2.1.4.

Algorithm A.1: *MapAlgorithm*: The top level algorithm of EagerMap.

```
    Input: commMatrixInit[][], nTasks
    Output: map[]
    LocalData: nElements, i, nGroups, rootGroup, commMatrix[][], groups[], previousGroups[]
    GlobalData: nLevels, execElInLevel[], hardwareTopologyRoot
 1  begin
 2  |   for i←0 ; i<nTasks ; i←i+1 do
 3  |   |   groups[i].id ← i;
 4  |   |   groups[i].nElements ← 0;
 5  |   end
 6  |   nElements ← nTasks;
 7  |   commMatrix ← commMatrixInit;
 8  |   for i←1 ; i<nLevels ; i←i+1 do
 9  |   |   previousGroups ← groups;
10  |   |   [nGroups, groups] ← GenerateGroupsForLevel(commMatrix, nElements, i, previousGroups,
    |   |   execElInLevel[i]);
11  |   |   if i < nLevels-1 then
12  |   |   |   commMatrix ← RecreateMatrix(commMatrix, groups, nGroups);
13  |   |   end
14  |   |   nElements ← nGroups;
15  |   end
16  |   rootGroup.nElements ← nElements;
17  |   for i←1 ; i<nElements ; i←i+1 do
18  |   |   rootGroup.elements[i] ← groups[i];
19  |   end
20  |   MapGroupsToTopology(archTopologyRoot, rootGroup, map);
21  |   return map;
22  end
```

A.2.1.2 Generating the Groups for a Level of the Architecture Hierarchy

The *GenerateGroupsForLevel* algorithm, described in Algorithm A.2, handles the creation of all groups for a given level of the architecture hierarchy. It expects that the levels of hierarchy up to the previous processed level to be already grouped, which are defined in *previousGroups*. The maximum number of groups is the number of hardware objects of that level. The selection of which elements belong to each group is performed by *GenerateGroup*.

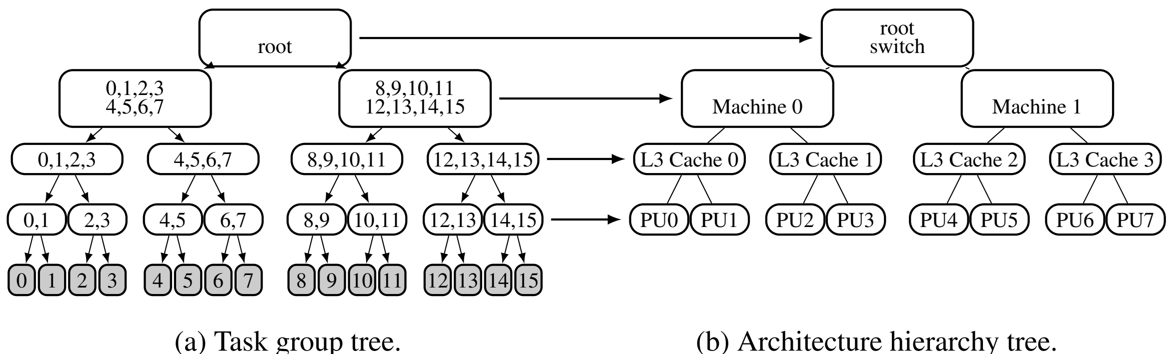
The *GenerateGroup* algorithm, in Algorithm A.3, groups elements that present a large amount of communication among themselves. For the grouping, the algorithm adopts a greedy strategy. The strategy works as follows. Each iteration of the loop in line 2 adds one element to the group. The added element, expressed by the *winner* variable, is the one that presents the largest amount of communication relative to the elements already in the group. The *chosen* variable is used to avoid selecting the same element more than once. *GenerateGroup* can be parallelized in the loop of line 4, where each thread would compute its local *winner* in parallel. After the loop, the master thread would select the *winner* among the ones calculated by each thread.

Algorithm A.2: $GenerateGroupsForLevel$: Generates the groups for a level of the architecture hierarchy.

```
   Input: commMatrix[][], nElements, level, previousGroups[], avlGroups
   Output: nGroups, groups[]
   LocalData: chosen[], elPerGroup, leftover, gi, inGroup, i, newGroup
 1 begin
 2     if nElements > avlGroups then
 3         nGroups ← avlGroups;
 4     else
 5         nGroups ← nElements;
 6     end
 7     elPerGroup ← nElements / nGroups;
 8     leftover ← nElements % nGroups;
 9     for i←0 ; i<nElements ; i←i+1 do
10         chosen[i] ← 0;
11     end
12     gi ← 0;
13     for i←0 ; i<nElements ; i←i+inGroup do
14         inGroup ← elPerGroup;
15         if leftover > 0 then
16             inGroup ← inGroup + 1;
17             leftover ← leftover - 1;
18         end
19         newGroup ← GenerateGroup(commMatrix, nElements, inGroup, chosen, previousGroups);
20         newGroup.nElements ← inGroup;
21         newGroup.id ← gi;
22         groups[gi] ← newGroup;
23         gi ← gi + 1;
24     end
25     return [nGroups, groups];
26 end
```

A.2.1.3 Computing the Communication Matrix for the next Level of the Architecture Hierarchy

$RecreateMatrix$, described in Algorithm A.4, regenerates the communication matrix to be used for the next level of the architecture hierarchy. The new communication matrix has an order of $nGroups$. It contains the amount of communication between the groups. It is calculated by



(a) Task group tree. (b) Architecture hierarchy tree.

Figure A.2: Mapping 16 tasks in an architecture with 8 PUs, L3 caches shared by 2 PUs, 2 processors per machine, and 2 machines.

Algorithm A.3: *GenerateGroup*: Generates one group of elements that communicate.

```
   Input: commMatrix[][], totalElements, groupElements, chosen[], previousGroups[]
   Output: group
   LocalData: i, j, w, wMax, winners[], winner
 1 begin
 2   for i←0 ; i<groupElements ; i←i+1 do
 3     wMax ← -1;
 4     for j←0 ; j<totalElements ; j←j+1 do
 5       if chosen[j]=0 then
 6         w ← 0;
 7         for k←0 ; k<i ; k←k+1 do
 8           w ← w + commMatrix[j][winners[k]];
 9         end
10         if w > wMax then
11           wMax ← w;
12           winner ← j;
13         end
14       end
15     end
16     chosen[winner] ← 1;
17     winners[i] ← winner;
18     group.elements[i] ← previousGroups[winner];
19   end
20   return group
21 end
```

Algorithm A.4: *RecreateMatrix*: Calculates the communication matrix for the next level.

```
   Input: commMatrix, groups[], nGroups
   Output: newCommMatrix[][]
   LocalData: i, j, k, z, w
 1 begin
 2   for i←0 ; i<nGroups-1 ; i←i+1 do
 3     for j←i+1 ; j<nGroups ; j←j+1 do
 4       w ← 0;
 5       for k←0 ; k<groups[i].nElements; k←k+1 do
 6         for z←0 ; z<groups[j].nElements; z←z+1 do
 7           w ← w + commMatrix[ groups[i].elements[k].id ][ groups[j].elements[z].id ];
 8         end
 9       end
10       newCommMatrix[i][j] ← w;
11       newCommMatrix[j][i] ← w;
12     end
13   end
14   return newCommMatrix;
15 end
```

summing up the amount of communication between the elements of different groups.

Algorithm A.5: $MapGroupsToTopology$: Maps the group tree to the hardware topology tree.

```
   Input: hardwareObj, group, map[]
   LocalData: i
1  begin
2      if hardwareObj.type = ProcessingUnit then
3          for i←0 ; i<group.nElements ; i←i+1 do
4              map[ group.elements[i].id ] ← hardwareObj.id;
5          end
6      else if hardwareObj.nSharers > 1 then
7          for i←0; i<group.nElements; i←i+1 do
8              MapGroupsToTopology(hardwareObj.linked[i], group.elements[i], map);
9          end
10     else
11         MapGroupsToTopology(hardwareObj.linked[0], group, map);
12     end
13 end
```

A.2.1.4 Mapping the Group Tree to the Architecture Topology Tree

The algorithm to map the group tree to the architecture topology tree is $MapGroupsToTopology$, detailed in Algorithm A.5. It is a recursive algorithm that performs a recursion for each level of the architecture hierarchy. The stop condition of the recursion is when it reaches the lowest level of the architecture topology, the processing unit (line 2). If the stop condition is not fulfilled, the algorithm already knows that the maximum number of groups per level never exceeds the number of hardware objects of that level, as explained in $GenerateGroupsForLevel$. Therefore, if the level of the architecture hierarchy in the recursion is shared (line 6), the algorithm only assigns one hardware object of the following level to each group and and recursively calls itself for each element of the following level. Otherwise, the level is private and is not considered for mapping (line 10).

A.2.2 Mapping Example in a Multi-level Hierarchy

For a better understanding of how our mapping algorithm works, we present an example of mapping 16 tasks in a cluster of 2 machines, each consisting of 8 cores, with L3 caches shared by 2 cores and 2 processors per machine. Figure A.2b illustrates the hierarchy of the machine. Private levels, with arity 1, such as L1 and L2 caches, as well as the processor, are not relevant. The global variable $nLevels$ has the value of 4, since there are 4 levels in the hierarchy. Therefore, $execElInLevel$ has 4 positions. Position 0 represents the tasks, its value is not used. Position 1 represents the processing units (the cores in this case), its value is 8. Position 2 represents the L3 caches, its value is 4. Finally, position 3 represents the machines, its value is 2.

Regarding Algorithm A.1 ($MapAlgorithm$), the loop in line 8 is executed 3 times. In the

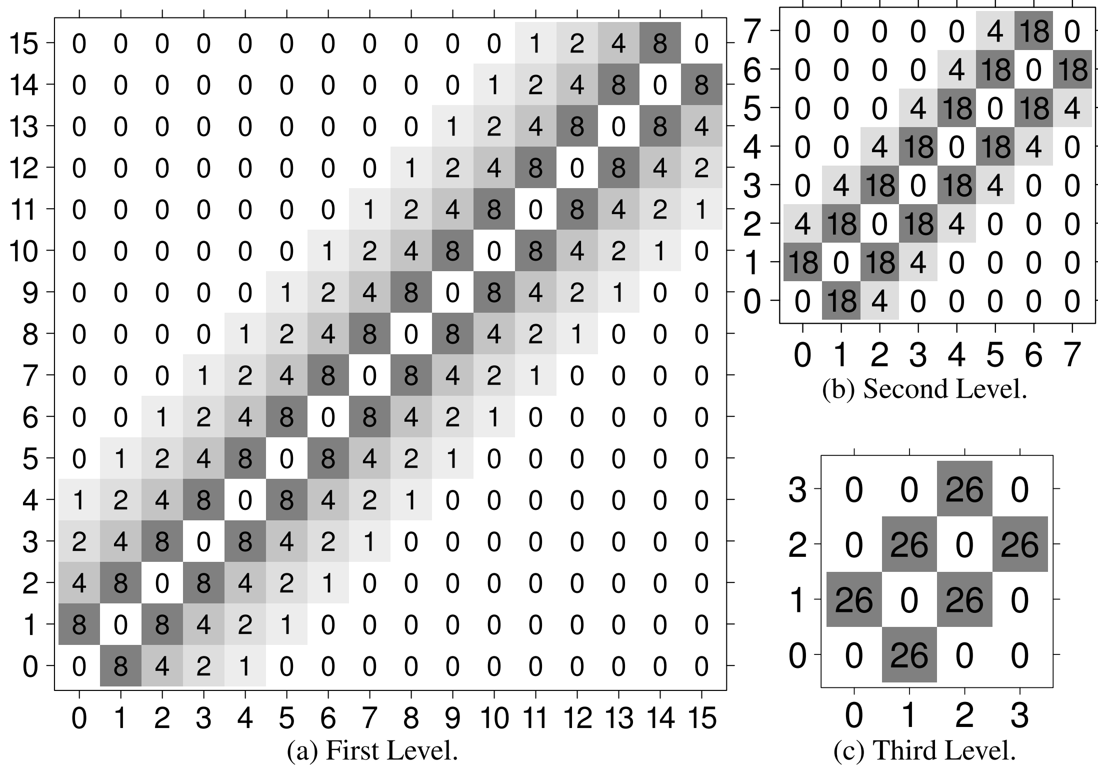


Figure A.3: Communication matrices used in our example.

i	winners		winner	wmax	w_0	w_1	w_2	w_3	w_4	w_{5-15}
0			0	0	0	0	0	0	0	0
1	0		1	8	-	8	4	2	1	0
2	0	1	-	-	-	-	-	-	-	-

Figure A.4: Example of the $GenerateGroup$ algorithm, when generating the first group in our example.

first iteration, it generates the groups of tasks that execute in each core. Since the number of tasks is 16 and the number of cores is 8, it generates 8 groups of 2 tasks each. The communication matrix used in the first iteration is shown in Figure A.3a. We demonstrate how the first group is generated in Figure A.4 ($GenerateGroup$, Algorithm A.3). When i is 0, the $winners$ vector is empty and task 0 is selected to be the first one of the group. When i is 1, the task that presents the most amount of communication to the tasks in the $winners$ vector is task 1. Therefore, task 1 is selected as $winner$ and enters the group. Algorithm $GenerateGroup$ then returns a group formed by tasks [0, 1], implicitly stored in the $group.elements$ vector.

$GenerateGroupsForLevel$ (Algorithm A.2) repeats the same procedure until all groups of the level are formed. Then, the communication matrix for the next level, in Figure A.3b, is calculated by $RecreateMatrix$ (Algorithm A.4). The second iteration of the loop in line 8 of $MapAlgorithm$ behaves similarly, but selects which groups of tasks from the previous iteration share each L3 cache. Since the number of groups was 8 and there are 4 L3 caches, it generates 4 groups of groups of tasks with 2 elements in each group.

In the third iteration of the loop in line 8 of $MapAlgorithm$, it selects which groups of groups of tasks share each machine. The communication matrix illustrated in Figure A.3c is used. Since the number of groups generated in the previous iteration was 4 and there are 2 machines, it generates 2 groups of groups of groups of tasks, 2 elements in each group. To complete the group tree, the $rootGroup$ variable points to these 2 groups of groups of groups of tasks. Afterwards, in line 19 of $MapAlgorithm$, the task group tree shown in Figure A.2a is finished.

$MapGroupsToTopology$ (Algorithm A.5) maps the task group vertices to vertices of the architecture hierarchy. The mapping begins from the root node. Tasks $[0-7]$ are mapped to Machine 0. Tasks $[0-3]$ are mapped to L3 cache 0. Tasks $[0-1]$ are mapped to PU 0 (core 0). The recursion continues until all groups of tasks are mapped to a PU (core). In this example, tasks x and $x+1$, where x is even, will be mapped to core $x/2$.

A.2.3 Complexity of the Algorithm

For the analysis of the complexity of the algorithm, we first introduce the variables used in the equations. E is the number of elements to be mapped in the current level of the architecture hierarchy (tasks or groups of tasks), which is different for each level. G is the number of elements per group. P is the number of processing units. N is the number of tasks to be mapped. L is the number of levels on the architecture hierarchy. To make it easier to calculate the complexity, we consider that $P \leq N$.

The complexity of $GenerateGroup$ is:

$$\sum_{i=1}^{G}\sum_{j=1}^{E} i = \sum_{i=1}^{G} E \cdot i \leq E \cdot G^2 \tag{A.1}$$

The complexity of $GenerateGroupsForLevel$ is shown in Equation A.2. The number of groups is $\frac{E}{G}$. Also, E is an upper bound for G.

$$\sum_{i=1}^{\frac{E}{G}} GenerateGroup \leq \sum_{i=1}^{\frac{E}{G}} E \cdot G^2 \leq \frac{E}{G} \cdot E \cdot G^2 \leq E^3 \tag{A.2}$$

The complexity of $RecreateMatrix$ is $O(E^2)$. The complexity of $MapGroupsToTopology$ is the same as performing a depth-first search, $O(V + C)$, where V is the number of vertices and C is the number of edges. Since our $groups$ variable implicitly forms a tree, we know that $C = V - 1$. The last level of this tree has N vertices, and the penultimate level has P vertices. The number of vertices of the other levels is the number of the previous level, divided by the number of sharers. The number of sharers is greater than 1 since we only keep track of shared levels. Therefore, $V \leq N + P + 2 \cdot P = O(N)$. With this, the complexity of $MapGroupsToTopology$ is $O(N)$.

The complexity of the top level algorithm, $MapAlgorithm$, depends on all previous algorithms, as shown in Equation A.3. To calculate the complexity, we have to take into account that the value of E changes on each level of the architecture hierarchy.

$$\begin{split}\sum_{i=0}^{L}\Big(GenerateGroupsForLevel + RecreateMatrix\Big) \\ +MapGroupsToTopology\end{split} \tag{A.3}$$

$$= \sum_{i=0}^{L}\Big(O\left(E_i^3\right) + O\left(E_i^2\right)\Big) + O(N) \tag{A.4}$$

We can rewrite Equation A.4 as Equation A.5 by considering that the algorithm iterates $L+1$ times (L levels of the architecture topology plus one level for the tasks) and that only shared topology levels are represented, which means that, in the worst case scenario, the topology will be a binary tree with P leaves and a maximum number of vertices equal to $2P-1$.

$$\leq \sum_{i=1}^{L}\left[O\left(\left(\frac{P}{2^{i-1}}\right)^3\right) + O\left(\left(\frac{P}{2^{i-1}}\right)^2\right)\right] + \big(O(N^3) + O(N^2)\big) + O(N) \tag{A.5}$$

$$\begin{split}\leq 2\big(O(P^3) + O(P^2)\big) + \big(O(N^3) + O(N^2)\big) + O(N) \\ = 3\big(O(N^3) + O(N^2)\big) + O(N) = O\left(N^3\right)\end{split} \tag{A.6}$$

With the simplifications shown in Equation A.6, we find that EagerMap has a polynomial complexity of $O(N^3)$.

A.3 Evaluation of EagerMap

In this section, we first show how we evaluate our proposal. Afterwards, we present the results obtained on its empirical evaluation.

A.3.1 Methodology of the Mapping Comparison

We discuss the benchmarks and architecture used in our evaluation, how we obtain the communication matrices, and other mapping strategies to which we compare EagerMap.

A.3.1.1 Benchmarks

For the evaluation of the mapping algorithms, we use applications from the MPI implementation of the NAS parallel benchmarks (NPB) (BAILEY et al., 1991), the OpenMP NPB

implementation (JIN; FRUMKIN; YAN, 1999), the High Performance Computing Challenge benchmark (LUSZCZEK et al., 2005) and the PARSEC benchmark suite (BIENIA et al., 2008). The NAS benchmarks were executed with the *B* input size, HPCC with an input matrix with 4000^2 elements and PARSEC was executed with the *native* input size.

A.3.1.2 Generating the Communication Matrices

For the MPI based benchmarks, we used the eztrace framework (TRAHAY et al., 2011) to trace all MPI messages sent by tasks and built a communication matrix based on the number of messages sent between tasks. For the benchmarks that use shared memory for implicit communication using memory accesses, we built a memory tracer based on the Pin binary analysis tool (LUK et al., 2005), similarly to (BARROW-WILLIAMS; FENSCH; MOORE, 2009). This tool traces all memory accesses from the tasks. We build a communication matrix by comparing memory accesses to the same memory address from different tasks and increment the matrix on every match.

A.3.1.3 Hardware Architecture

The hardware architecture used in our experiments is illustrated in Figure A.5, with 4 Intel Xeon X7550 processors for a total of 64 PUs. In this architecture, there are three possibilities for communication between tasks. Tasks running on the same core (case Ⓐ) can communicate through the fast L1 or L2 caches and have the highest communication performance. Tasks that run on different cores (case Ⓑ) have to communicate through the slower L3 cache, but can still benefit from the fast intra-chip interconnection. When tasks communicate across physical processors (case Ⓒ), they need to use the slow inter-chip interconnection. Hence, the communication performance in case Ⓒ is the slowest in this architecture.

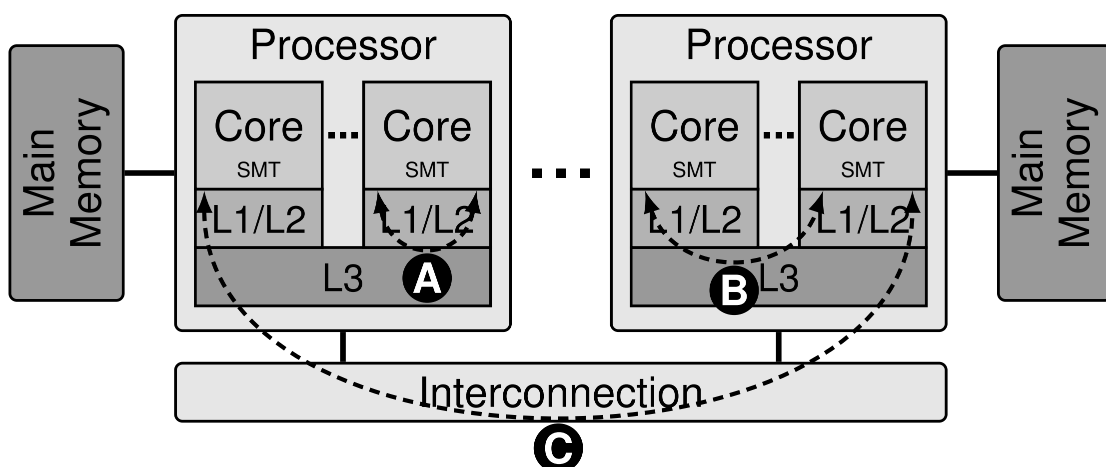


Figure A.5: Hardware architecture used in our experiments.

A.3.1.4 Comparison

We compare EagerMap to (i) a Random mapping, which is an average result of 30 different random mappings; (ii) TreeMatch (JEANNOT; MERCIER; TESSIER, 2014) version $0.2.3$; (iii) Scotch (PELLEGRINI, 1994) version 6.0; and (iv) MPIPP (CHEN et al., 2006).

A.3.2 Results

We evaluate the performance and quality of our algorithm, its stability, as well as the performance improvements obtained when mapping the tasks of the applications.

A.3.2.1 Performance of the Mapping Algorithms

Figure A.6 shows the execution time of the four mapping algorithms for all benchmarks. For 128 tasks, EagerMap is about 10 times faster than Scotch, 1000 times faster than TreeMatch, and 100,000 times faster than MPIPP. TreeMatch has an exponential complexity, so a much higher execution time is expected for it. Due to this, the difference in execution time between TreeMatch and EagerMap increases together with the number of tasks. MPIPP is much slower than the other mechanisms because it needs to perform several iterations to refine its initial random mapping. It did not finish executing when the number of tasks was higher than 128.

A.3.2.2 Quality of the Mapping

The quality of the calculated mapping determines the benefits that can be achieved. Quality is measured by the amount of locality achieved. We use Equation A.7 to calculate the quality, which is provided in the source code of TreeMatch. In this equation, n is the number of tasks, $M[i][j]$ is the amount of communication between tasks i and j, $map[x]$ is the processing unit

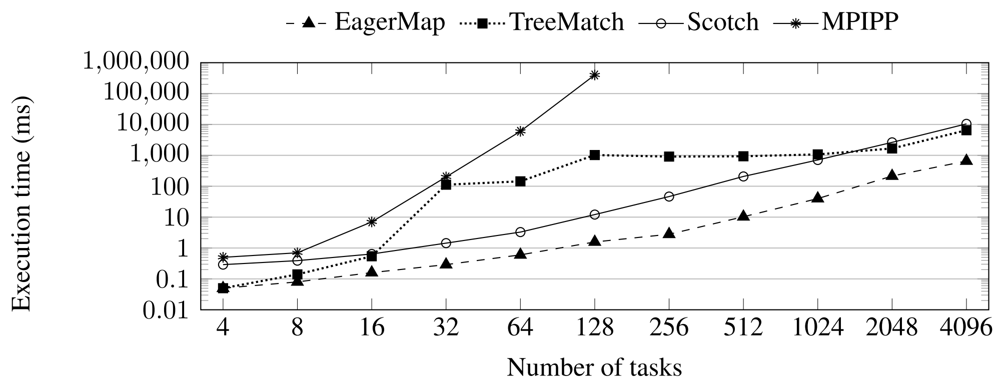


Figure A.6: Execution time (in ms) of the mapping algorithms, for different numbers of tasks to be mapped. Lower results are better.

(PU) mapped, and $lat[a][b]$ is the latency of the PUs in the hierarchy. We calculated the latencies using LMbench (MCVOY; STAELIN, 1996).

$$MappingQuality = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{M[i][j]}{lat\big[map[i]\big]\big[map[j]\big]} \tag{A.7}$$

Figure A.7 presents the mapping quality results for the communication matrices previously illustrated in Figure A.1. Applications with more structured communication patterns presented the highest improvements, as expected. CG-MPI, LU-MPI and HPCC (Phase 7) presented the highest improvements in accuracy because their communication can be easily optimized by mapping neighboring tasks together. In BT-MPI and MG-MPI, near and distant tasks communicate, presenting more challenges for the mapping algorithm. This happens because if neighboring tasks are mapped together, the communication between distant tasks does not improve. Likewise, mapping distant tasks together does not improve communication between neighboring tasks. In applications with less structured patterns, such as Streamcluster, a lower improvement in accuracy is expected because the ratio of communication between tasks that communicate more and tasks that communicate less is lower. Vips is the only application with unstructured communication, such that no task mapping was able to optimize its communication.

The quality obtained with MPIPP is similar to the random mapping, since MPIPP is based on refining an initial random mapping. Although MPIPP can work with few tasks, when the number of tasks increases, the possibilities of permutation are endless, and due to this it is more difficult to find new combinations to improve the initial solution. EagerMap, TreeMatch and Scotch presented similar results for all applications. This result demonstrates that EagerMap is able to achieve results as good as more complex algorithms due to the characteristics of the communication patterns we observed in Section A.2.

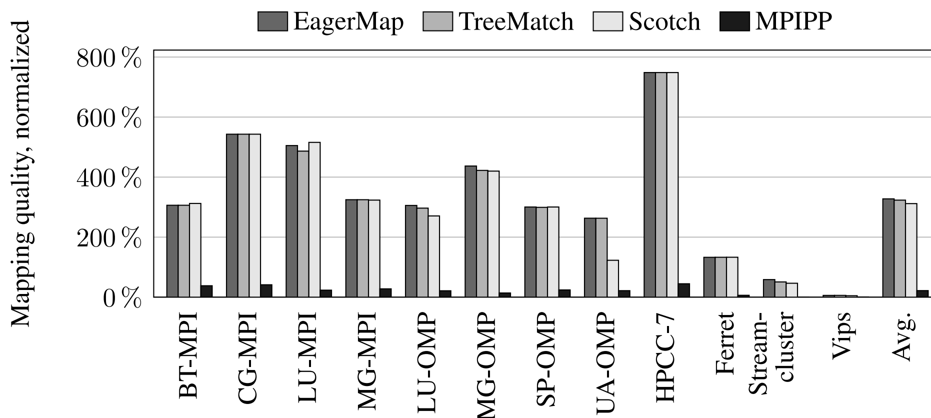


Figure A.7: Comparison of the mapping quality, normalized to the random mapping. Higher values are better.

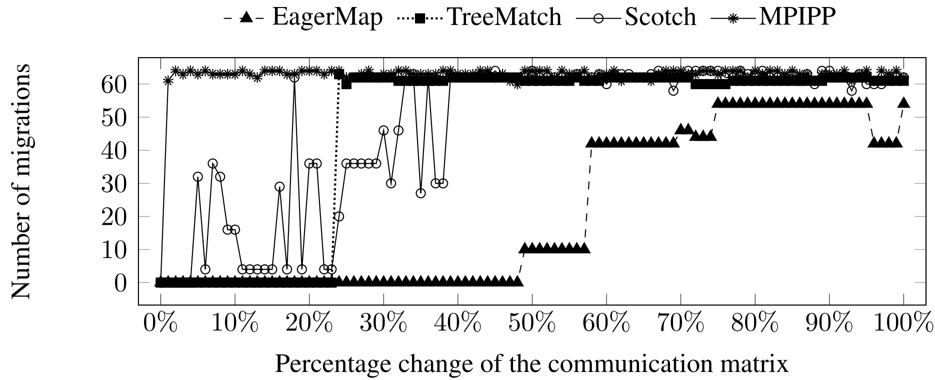


Figure A.8: Comparison of the mapping stability. Number of migrations depending on the percentile change of the communication matrix for an application consisting of 64 tasks. Less migrations are better.

A.3.2.3 Mapping Stability

Another important metric to consider for mapping algorithms is the numbers of migrations performed due to changes in the communication matrix, which we call stability. A high stability is important for online mapping techniques. In algorithms with low stability, small changes in the communication lead to unnecessary task migrations. When the stability of the algorithm is higher, more differences in the communication matrix are required to migrate tasks.

The stability results are shown in Figure A.8 for the BT-MPI benchmark running with 64 tasks. The x-axis corresponds to the percentage difference of a matrix relative to a base matrix. A value of $k\%$ indicates that all values of the matrix were changed randomly up to a limit of $k\%$ of the maximum of the matrix. Since the communication pattern itself does not change, ideally there should be no migrations. The y-axis shows the number of task migrations resulting from the changes. Lower numbers of migrations are better. The results show that our proposed algorithm has a higher stability than previous mechanisms. In this way, EagerMap introduces less overhead than the other algorithms, since they introduce unnecessary task migrations.

A.3.2.4 Performance Improvements

We executed applications using the mapping obtained with the algorithms. The execution time results for the MPI and OpenMP NAS Benchmarks are shown in Figure A.9. For these applications, since their communication pattern is stable, we calculated the mapping statically. As a case study for online mapping, we used the HPCC Benchmark, shown in Figure A.10, since it contains 16 phases with different communication behaviors. For each phase, we call the mapping algorithm and migrate tasks accordingly. Execution time results correspond to the average of 30 executions, and are normalized to the execution time of the original policy of the operating system. The confidence interval represented less than 1% for all algorithms.

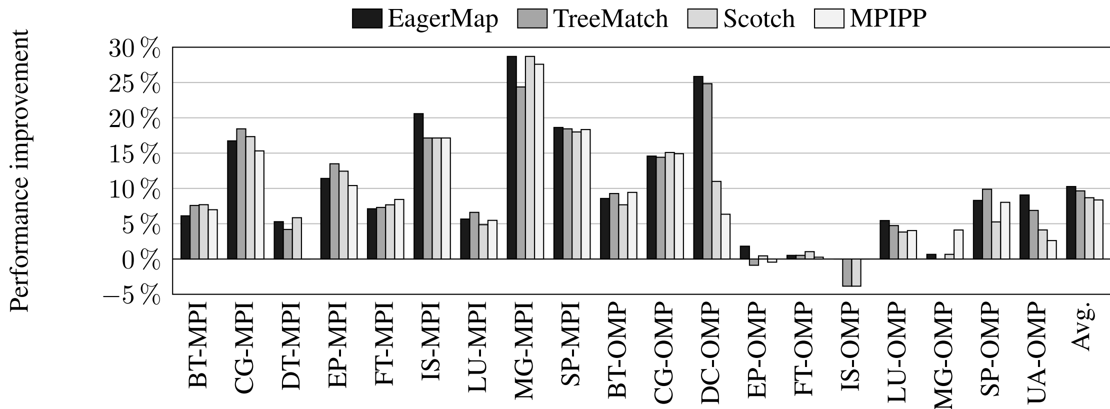


Figure A.9: Performance improvement compared to the OS mapping of the NAS-MPI and NAS-OMP benchmarks. Higher results are better.

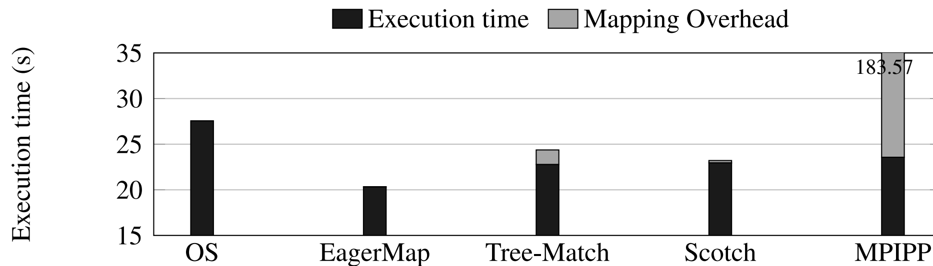


Figure A.10: Execution time of HPCC using online mapping. Lower results are better.

In general, applications with more structured communication patterns present higher performance improvements. For instance, CG-MPI's performance was significantly improved when compared to the OS mapping, since its communication pattern shows high potential for mapping, as discussed in Section A.3.2.2. On average, EagerMap improved performance by 10.3%, while TreeMatch, Scotch and MPIPP showed improvements of 9.6%, 8.7% and 8.4%, respectively. The results of the HPCC Benchmark (Figure A.10) show a larger improvement by EagerMap than related work. Also, the lower overhead of our algorithm does not harm the application performance as much as TreeMatch and MPIPP.

A.4 Conclusions

In parallel applications, the mapping of tasks to cores influences the communication performance. By mapping tasks that communicate to cores nearby in the memory hierarchy or to the same node in clusters or grids, performance is improved by making use of faster interconnections. The mapping algorithm selects which core will execute each task and plays a key role in this type of mapping.

In this thesis, we proposed a new mapping algorithm, EagerMap. In contrast to previous work, it adopts a more efficient method to select which tasks should be mapped together, based

on an analysis of the communication pattern of parallel applications. We performed experiments with a large set of benchmarks with different communication characteristics. Results show that EagerMap calculated better task mappings than the state-of-the-art, with a drastically lower overhead and better scaling. Furthermore, its high stability makes EagerMap more suitable for online mapping.

For the future, we will extend EagerMap to support arbitrary hardware hierarchies, not only those based on trees. EagerMap is licensed under the GPL and is available at `<http://github.com/ehmcruz/eagermap>`.

APPENDIX B SUMMARY IN PORTUGUESE

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

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

B.1 Introdução

O paralelismo a nível de threads tem aumentado em arquiteturas modernas devido ao maior número de núcleos por processador e processadores por sistema. Devido a isto, a complexidade das hierarquias de memória também aumentam. Tais hierarquias de memória podem incluir vários níveis de cache privadas ou compartilhadas, bem como tempos de acesso não uniforme a bancos de memória (arquiteturas NUMA). Um desafio importante para essas arquiteturas é a movimentação de dados entre os núcleos, caches e bancos de memória, que ocorre quando um núcleo realiza uma operação de memória. Neste contexto, a redução da movimentação de dados é uma meta importante para arquiteturas futuras para manter dimensionamento desempenho e diminuir o consumo de energia (BORKAR; CHIEN, 2011; COTEUS et al., 2011). Uma das soluções para reduzir o movimento de dados é melhorar a localidade do acesso à memória através do mapeamento de threads e dados (TORRELLAS, 2009; FELIU et al., 2012).

Mecanismos de mapeamento presentes no estado-da-arte tentam aumentar localidade mantendo threads que compartilham um grande volume de dados próximos na hierarquia de memória (*mapeamento de threads*), e mantendo os dados que as threads acessam próximos destas threads (*mapeamento de dados*). O desempenho e eficiência energética são aumentados por três razões principais. Em primeiro lugar, as faltas de cache são reduzidas, diminuindo o número de invalidações de linha de cache em dados compartilhado (MARTIN; HILL; SORIN, 2012) e reduzindo a duplicação de linhas de cache em vários caches (CHISHTI; POWELL; VIJAYKUMAR, 2005). Em segundo lugar, a localidade dos acessos à memória é aumentada através do mapeamento de dados para o nó NUMA onde é mais acessado *Diener2014, Ribeiro2009*. Em terceiro lugar, a utilização das interconexões no sistema é melhorada pela redução do tráfego em interconexões lentas entre os processadores, utilizando interconexões dentro dos processadores, mais eficientes. A falha em identificar o padrão de acesso à memória de uma aplicação pode levar a mapeamentos ruins que reduzem o desempenho e eficiência energética.

Nesta tese de doutorado, são propostos mecanismos para realizar o mapeamento de threads e dados a fim de se melhorar a localidade dos acessos à memória. Os mecanismos utilizam a infraestrutura da unidade de gerência de memória dos processadores (MMU), já que todos os acessos à memória são realizados pela mesma. Os mecanismos permitem que o padrão de acesso à memória seja identificado dinamicamente (durante a execução da aplicação). O sistema

operacional, também dinamicamente, utiliza as informações geradas pelos mecanismos e realiza o mapeamento de trhreads e dados. A precisão dos mecanismos aqui propostos é maior do que a de técnicas de trabalhos relacionados, além de apresentarem uma menor sobrecarga. Isto acontece porque os mecanismos aqui propostos são híbridos, tendo um suporte em hardware, que permite um monitoramento dos acessos à memória de forma transparente.

Este capítulo está organizado da seguinte forma. A Seção B.2 analiza o mapeamento e estado-da-arte em maiores detalhes. A Seção B.3 explica os mecanismos propostos nesta tese. A Seção B.4 descreve os experimentos. A Seção B.5 expõe as conclusões e trabalhos futuros.

B.2 Visão Geral Sobre Mapeamento Baseado em Compartilhamento

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 compartilhamento de dados das aplicações, que tem uma dificuldade variável dependendo do paradigma de programação paralela adotado. Quando utilizado paradigmas de programação orientados a passagem de mensagens, a descoberta do padrão de troca de dados é menos desafiadora, já que a troca é explícita, sendo feita através de chamadas à funções. Entretanto, com o uso de programação paralela para memória compartilhada em ambientes multiprocessados, a tarefa de descoberta do padrão de troca de dados se torna mais difícil, pois a troca é implícita e ocorre através do acesso a regiões de memória compartilhadas por diferentes threads. O foco dos mecanismos propostos nesta tese são aplicações baseadas no paradigma de memória compartilhada.

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. Além disso, o tempo para dois núcleos dentro de um mesmo processador 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, é reduzida a sobrecarga imposta por protocolos de coerência de cache.

Em relação à arquiteturas com características de acesso não-uniforme à memória (NUMA), além de mapeamento de thread, também é importante mapear os dados. 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 propos-

tos. Algumas técnicas exigem etapas custosas de *profiling*, desencorajando o uso das mesmas. Outras necessitam que os programadores das aplicações insiram anotações no código fonte ou, aumentando a complexidade da programação. 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 pode ser estático ou dinâmico. No mapeamento estático, análises prévias podem ser feitas com a aplicação. Etapas custosas, como simulação, são utilizadas para monitorar os acessos à memória e descobrir o padrão de compartilhamento. 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.

Fazendo uma análise do estado-da-arte, a maioria dos trabalhos realiza mapeamento de threads ou dados apenas, mas não os dois juntos. No caso de mecanismos de mapeamento de threads, eles não são capazes de reduzir a quantidade de acessos remotos em arquiteturas NUMA. Por outro lado, os mecanismos de mapeamento de dados não são capazes de reduzir faltas de cache ou lidar corretamente com o mapeamento de páginas compartilhadas por múltiplas threads. A maioria dos mecanismos que realizam ambos os mapeamentos têm várias desvantagens. Cruz et al. (2011) é um mecanismo estático que depende da informação de execuções anteriores, sendo limitados a aplicações cujo comportamento não mudam entre execuções ou durante a execução. ForrestGOMP (BROQUEDIS et al., 2010a) é dinâmico, mas exige anotações de código fonte por parte do programador para funcionar corretamente. kMAF (DIENER et al., 2014) usa amostragem e tem uma alta sobrecarga quando aumentamos a quantidade de amostras para alcançar uma maior precisão. Outros mecanismos dependem indiretamente estatísticas obtidas por contadores de hardware, que não representam com precisão o comportamento de acesso à memória de aplicações paralelas. Várias propostas exigem arquiteturas específicas, APIs ou linguagens de programação para trabalhar, o que limitam a sua aplicabilidade.

Com esta análise do estado-da-arte, podemos concluir que, atualmente, não existe um mecanismo dinâmico que pode ser aplicada a qualquer aplicação baseada em memória compartilhada, em que aumentar a sua precisão não aumenta não drasticamente a sua sobrecarga. As propostas desta tese tentam preencher esta lacuna. Elas realizam o mapeamento tanto de threads como de dados, otimizando o uso de cache e diminuindo acessos remotos e uso de interconexões entre processadores. As propostas são implementadas diretamente na MMU da arquitetura, o que permite monitorar muito mais acessos à memória do que os trabalhos relacionados de uma forma não intrusiva. Desta forma, pode-se conseguir uma precisão muito maior nos padrões detectados, enquanto mantendo uma baixa sobrecarga.

B.3 Mecanismos Propostos para Mapeamento Baseado em Compartilhamento

Todas as propostas desta tese são implementadas na MMU. A MMU de processadores atuais que suportam memória virtual traduz endereços de memória virtual para endereços de memória física em todos os acessos de memória. Em cada acesso à memória, a MMU verifica se a página tem uma entrada válida na TLB. Se isso acontecer, o endereço virtual é traduzido para um endereço físico e o acesso à memória é executado. Se a entrada não estiver na TLB, a MMU evita uma entrada antiga, busca a entrada necessária na tabela de páginas e armazena a entrada na TLB antes de prosseguir com a tradução de endereços e acesso à memória. No resto desta seção, são apresentados os mecanismos propostos para realizar o mapeamento.

B.3.1 LAPT – Locality-Aware Page Table

A primeira proposta é um mecanismo chamado LAPT. O LAPT implementa mudanças no subsistema de memória virtual, nos níveis tanto no hardware como de software. No nível de hardware, o LAPT mantém o controle de todas as faltas na TLB, como mostrado na Figura B.1, detectando o compartilhamento de dados entre threads em uma matriz chamada Matriz de Compartilhamento, e registrando uma lista com as últimas threads que acessaram cada página. No nível de software, o sistema operacional mapeia as threads para núcleos com base na Matriz de Compartilhamento e mapeia as páginas de memória para nós NUMA verificando quais threads acessaram cada página.

B.3.2 SAMMU – Sharing-Aware Memory Management Unit

A segunda proposta é um mecanismo chamado SAMMU. Uma visão geral de alto nível do funcionamento da MMU, TLB e SAMMU é ilustrada na Figura B.2. A operação da MMU é

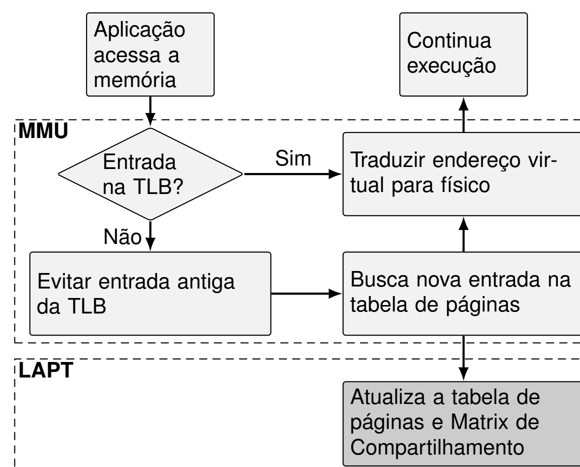


Figure B.1: Visão geral do comportamento da MMU e do LAPT.

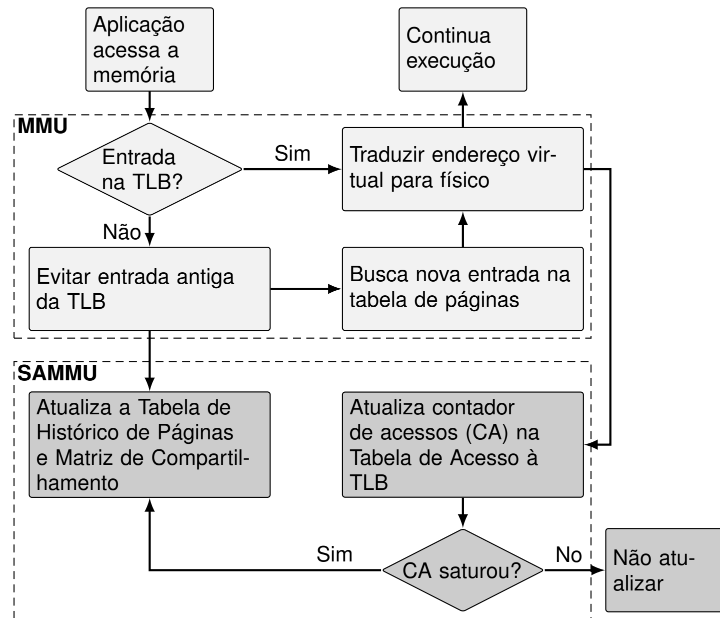


Figure B.2: Visão geral da MMU e do SAMMU.

estendida de duas maneiras, ocorrendo em paralelo com a operação normal do MMU sem parar de execução da aplicação:

1. O SAMMU conta o número de vezes que cada entrada TLB é acessado a partir do núcleo local. Isso permite a coleta de informações sobre as páginas acessadas por cada thread. Para tal, é armazenado um contador de acessoas (CA) por entrada da TLB, em uma tabela que chamamos de *Tabela de Acesso à TLB*, que é armazenada na MMU.
2. Em cada entrada evitada da TLB ou quando um contador CA satura, o SAMMU analisa estatísticas sobre a página e as armazena na memória primária em duas estruturas separadas. A primeira estrutura é a *Matriz de Compartilhamento*, que estima a quantidade de partilha entre as threads. A segunda estrutura é a *Tabela de Histórico de Páginas*, que contém uma entrada por página física no sistema, com informações sobre as threads e nós NUMA que os acessaram, e é indexado pelo endereço da página física.

B.3.3 IPM – Intense Pages Mapping

Uma visão geral de alto nível do funcionamento do MMU, TLB e do IPM é ilustrado na Figura B.3. Em cada falta na TLB, o IPM armazena o *time stamp* na memória principal em uma estrutura chamada *Tabela de Acesso à TLB*. Se uma entrada da TLB devem ser evitada para armazenar a nova entrada, o IPM carrega da Tabela de Acesso à TLB o *time stamp* correspondente à entrada TLB evitada. Subtraindo-se o *time stamp* carregado do *time stamp* atual, o IPM sabe quanto tempo a entrada evitada ficou armazenada na TLB. O IPM, em seguida, usa essa diferença de tempo para atualizar as informações contidas na *Matriz de Compartilhamento* e *Tabela de Histórico de Páginas*. Por último, o IPM notifica o sistema operacional se uma

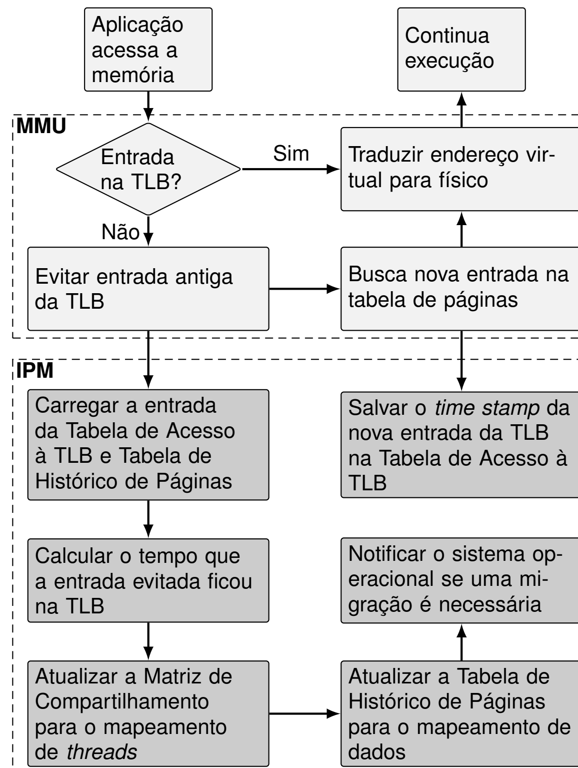


Figure B.3: Visão geral da MMU e do IPM.

migração página pode melhorar o desempenho.

B.4 Avaliação dos Mecanismos Propostos

Foram realizados experimentos em um simulador completo de sistema, uma máquina real com TLB gerida via *software* e duas máquinas reais com TLB gerida via *hardware*. O simulador completo de sistema utilizado foi o Simics (MAGNUSSON et al., 2002) estendido com o modelo de memória GEMS-Ruby (MARTIN et al., 2005) e do modelo de interconexão Garnet (AGARWAL et al., 2009), sendo simulado uma máquina com 8 núcleos organizados em 4 nós NUMA. A máquina real com TLB gerida via *software* é da arquitetura *Itanium/Montecito* e possui 8 núcleos distribuídos em 2 nós NUMA. As máquinas reais com TLB gerida via *hardware* são das arquiteturas *Nehalem* e *SandyBridge*, e e possuem 32/64 núcleos virtuais distribuídos em 2/4 nós NUMA, respectivamente. Tais máquinas serão referenciadas por *Xeon32* e *Xeon64*.

Como as propostas desta tese são extensões para o hardware atual, nas máquinas reais com TLB gerida via *hardware*, usamos o Pin (BACH et al., 2010), uma ferramenta de instrumentação de binários, para gerar informações sobre os mapeamentos de threads e de dados. Como as cargas de trabalho, utilizou-se a implementação OpenMP dos benchmarks paralelos do NAS (JIN; FRUMKIN; YAN, 1999) v3.3.1 e o conjunto de benchmarks do PARSEC (BIENIA et al., 2008) v3.0. Também utilizou-se uma aplicação real de simulação sísmica chamada

Ondes3D (DUPROS et al., 2008).

Os resultados obtidos no Simics, Itanium e Xeon64 são melhores do que os resultados obtidos na Xeon32. Como Simics e Xeon64 tem 4 nós NUMA (enquanto Xeon32 tem 2), a probabilidade de encontrar o nó correto para uma página sem qualquer conhecimento do padrão de acesso à memória é apenas 25% neles (enquanto ele é de 50% na Xeon32). Em relação à Itanium, embora também tenha apenas 2 nós NUMA, um mapeamento melhor tem um impacto maior, porque a latência de um acesso à memória remota é muito maior do que na Xeon32.

A maioria das aplicações são mais sensíveis ao mapeamento de dados do que ao mapeamento de threads, o que pode ser observado nos resultados pelo fato de que o tráfego interchip apresentou uma redução maior do que faltas de cache. Isso acontece porque, mesmo se um aplicativo não compartilha muitos dados entre suas threads, cada thread ainda terá de acessar seus próprios dados privados, o que só pode ser melhorado através de mapeamento de dados. É importante notar que isto não significa que o mapeamento dos dados é mais importante do que o mapeamento de threads, porque a eficácia de mapeamento de dados depende de mapeamento de threads para páginas compartilhadas.

As propostas desta tese apresentaram resultados semelhantes ao mapeamento *oracle*, demonstrando suas eficácias. Também teve um desempenho significativamente melhor do que o mapeamento aleatório para a maioria dos casos. Isto mostra que os ganhos em relação ao sistema operacional não são devidos às migrações desnecessárias introduzidas pelo sistema operacional, mas devido a uma utilização mais eficiente dos recursos. Em comparação à outras políticas de mapeamento do estado-da-arte, as nossas propostas apresentaram melhores resultados para a maioria das aplicações.

B.5 Conclusão e Trabalhos Futuros

Nesta tese de doutorado, propusemos novas soluções para aumentar desempenho utilizando o mapeamento de threads e dados. As soluções suprem uma lacuna do estado-da-arte: são capazes de atingir alta precisão com uma sobrecarga baixíssima. Para tal, as soluções fazem uso da MMU presentes em processadores atuais, já que a MMU já tem acesso a informações sobre todos os acessos à memória realizada por cada núcleo. O mecanismo mais simples que é proposto, o LAPT, rastreia as threads que acessam cada página e calcula a quantidade de páginas compartilhadas pelas threads. Também é proposto o SAMMU, que além do que o LAPT faz, estima a quantidade de acessos realizados a cada página de todos os nós NUMA. Finalmente, é proposto o IPM, que fornece informações similares às do SAMMU, mas tem um procedimento mais simples e utiliza informações sobre o tempo de cada página tem sua entrada na TLB, em vez de a quantidade de acessos à memória. Com o suporte de hardware proposto, os mecanismos aqui apresentados são capazes de oferecer uma alta precisão com uma baixa sobrecarga de desempenho.

Os mecanismos foram avaliados em três ambientes diferentes: um simulador de sistema

completo, uma máquina real com TLB gerida por *software* TLB e duas máquinas reais com TLB gerida por *hardware*. Os mecanismos proveram melhorias substanciais em todos os ambientes. Para exemplificar, no simulador completa de sistema, LAPT, SAMMU e IPM reduziram o tempo de execução por uma média de 10,1%, 10,9% e 11,4%, respectivamente. O ganho de desempenho ocorreu, principalmente, devido a uma redução substancial do número de faltas em cache e uso das interconexões entre processadores. Os mecanismos desta tese foram comparados com trabalhos relacionados, onde provou-se que os mecanismos aqui apresentados, que permitem atingir uma maior precisão com baixa sobrecarga, são capazes de fornecer melhor desempenho.

Como trabalho futuro, pretende-se avaliar e propor soluções para melhorar a localidade memória visando sistemas mais recentes, como o Xeon Phi, que também apresentam diferentes latências de acesso a dados entre os núcleos. Além disso, queremos expandir o mapeamento para considerar outros aspectos das arquiteturas, tais como o uso de unidades funcionais e balanceamento de carga. Outra possibilidade é a de considerar o mapeamento em computação heterogênea, a fim de identificar as características de uma aplicação paralela e mapear as threads para os núcleos que melhor se adaptem às suas características.

REFERENCES

AGARWAL, N. et al. GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator. In: **IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.: s.n.], 2009. p. 33–42.

AMD. **AMD Opteron™ 6300 Series processor Quick Reference Guide**. [S.l.], 2012.

AWASTHI, M. et al. Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers. In: **Parallel Architectures and Compilation Techniques (PACT)**. [S.l.: s.n.], 2010. p. 319–330.

AZIMI, R. et al. Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring. **ACM SIGOPS Operating Systems Review**, v. 43, n. 2, p. 56–65, abr. 2009.

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

BAILEY, D. H. et al. The NAS Parallel Benchmarks. **International Journal of High Performance Computing Applications**, v. 5, n. 3, p. 66–73, 1991.

BARROW-WILLIAMS, N.; FENSCH, C.; MOORE, S. A Communication Characterisation of Splash-2 and Parsec. In: **IEEE International Symposium on Workload Characterization (IISWC)**. [S.l.: s.n.], 2009. p. 86–97.

BELLARD, F. Qemu, a fast and portable dynamic translator. In: **USENIX Annual Technical Conference (ATEC)**. Berkeley, CA, USA: USENIX Association, 2005. p. 41–41.

BIENIA, C.; KUMAR, S.; LI, K. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on Chip-Multiprocessors. In: **IEEE International Symposium on Workload Characterization (IISWC)**. [S.l.: s.n.], 2008. p. 47–56.

BIENIA, C. et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In: **International Conference on Parallel Architectures and Compilation Techniques (PACT)**. [S.l.: s.n.], 2008. p. 72–81.

BINKERT, N. et al. The gem5 simulator. **ACM SIGARCH Computer Architecture News**, ACM, v. 39, n. 2, p. 1–7, 2011.

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

BORKAR, S.; CHIEN, A. A. The Future of Microprocessors. **Communications of the ACM**, v. 54, n. 5, p. 67–77, 2011.

BRANDFASS, B.; ALRUTZ, T.; GERHOLD, T. Rank reordering for MPI communication optimization. **Computers & Fluids**, Elsevier Ltd, p. 372–380, jan. 2012.

BROQUEDIS, F. et al. Structuring the execution of OpenMP applications for multicore architectures. In: **IEEE International Parallel & Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2010. p. 1–10.

BROQUEDIS, F. et al. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In: **Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)**. [S.l.: s.n.], 2010. p. 180–186.

CABEZAS, V. C.; STANLEY-MARBELL, P. Parallelism and data movement characterization of contemporary application classes. In: **ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)**. [S.l.: s.n.], 2011.

CASAVANT, T. L.; KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. **IEEE Transactions on Software Engineering**, IEEE Press, v. 14, n. 2, p. 141–154, feb. 1988.

CHEN, H. et al. MPIPP: An Automatic Profile-guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters. In: **International Conference on Supercomputing (SC)**. [S.l.: s.n.], 2006. p. 353–360.

CHISHTI, Z.; POWELL, M. D.; VIJAYKUMAR, T. N. Optimizing Replication, Communication, and Capacity Allocation in CMPs. **ACM SIGARCH Computer Architecture News**, v. 33, n. 2, p. 357–368, may 2005.

CORBET, J. **AutoNUMA: the other approach to NUMA scheduling**. 2012. Available from Internet: <http://lwn.net/Articles/488709/>.

CORBET, J. **Toward better NUMA scheduling**. 2012. Available from Internet: <http://lwn.net/Articles/486858/>.

COTEUS, P. W. et al. Technologies for exascale systems. **IBM Journal of Research and Development**, v. 55, n. 5, p. 14:1–14:12, sep. 2011.

CRUZ, E. et al. Using memory access traces to map threads and data on hierarchical multi-core platforms. In: **IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)**. [S.l.: s.n.], 2011.

CRUZ, E. H. M.; ALVES, M. A. Z.; NAVAUX, P. O. A. Process Mapping Based on Memory Access Traces. In: **Symposium on Computing Systems (WSCAD-SCC)**. [S.l.: s.n.], 2010. p. 72–79.

CRUZ, E. H. M. et al. Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols. **Journal of Parallel and Distributed Computing (JPDC)**, v. 74, n. 3, p. 2215–2228, mar. 2014.

CRUZ, E. H. M. et al. Optimizing Memory Locality Using a Locality-Aware Page Table. In: **International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. [S.l.: s.n.], 2014. p. 198–205.

CRUZ, E. H. M.; DIENER, M.; NAVAUX, P. O. A. Using the Translation Lookaside Buffer to Map Threads in Parallel Applications Based on Shared Memory. In: **IEEE International Parallel & Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2012. p. 532–543.

CRUZ, E. H. M.; DIENER, M.; NAVAUX, P. O. A. Communication-Aware Thread Mapping Using the Translation Lookaside Buffer. **Concurrency Computation: Practice and Experience**, v. 22, n. 6, p. 685–701, 2015.

CRUZ, E. H. M. et al. An Efficient Algorithm for Communication-Based Task Mapping. In: **International Conference on Parallel, Distributed, and Network-Based Processing (PDP)**. [S.l.: s.n.], 2015. p. 207–214.

CUESTA, B. et al. Increasing the Effectiveness of Directory Caches by Avoiding the Tracking of Non-Coherent Memory Blocks. **IEEE Transactions on Computers**, v. 62, n. 3, p. 482–495, 2013.

DASHTI, M. et al. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In: **Architectural Support for Programming Languages and Operating Systems (ASPLOS)**. [S.l.: s.n.], 2013. p. 381–393.

DEVINE, K. D. et al. Parallel hypergraph partitioning for scientific computing. In: **IEEE International Parallel & Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2006. p. 124–133.

DIENER, M.; CRUZ, E. H. M.; NAVAUX, P. O. A. Communication-Based Mapping Using Shared Pages. In: **IEEE International Parallel & Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2013. p. 700–711.

DIENER, M. et al. kMAF: Automatic Kernel-Level Management of Thread and Data Affinity. In: **International Conference on Parallel Architectures and Compilation Techniques (PACT)**. [S.l.: s.n.], 2014. p. 277–288.

DIENER, M. et al. Communication-Aware Process and Thread Mapping Using Online Communication Detection. **Parallel Computing**, v. 43, n. March, p. 43–63, 2015.

DIENER, M. et al. Characterizing Communication and Page Usage of Parallel Applications for Thread and Data Mapping. **Performance Evaluation**, v. 88-89, n. June, p. 18–36, 2015.

DIENER, M. et al. Evaluating Thread Placement Based on Memory Access Patterns for Multi-core Processors. In: **IEEE International Conference on High Performance Computing and Communications (HPCC)**. [S.l.: s.n.], 2010. p. 491–496.

DUPROS, F. et al. Exploiting Intensive Multithreading for the Efficient Simulation of 3D Seismic Wave Propagation. In: **IEEE International Conference on Computational Science and Engineering (CSE)**. [S.l.: s.n.], 2008. p. 253–260.

DUPROS, F. et al. Parallel simulations of seismic wave propagation on NUMA architectures. In: **Parallel Computing: From Multicores and GPU's to Petascale**. [S.l.: s.n.], 2010. p. 67–74.

FELIU, J. et al. Understanding Cache Hierarchy Contention in CMPs to Improve Job Scheduling. In: **International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2012.

GABRIEL, E. et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In: **Recent Advances in Parallel Virtual Machine and Message Passing Interface**. [S.l.: s.n.], 2004.

HOEFLER, T.; SNIR, M. Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In: **International Conference on Supercomputing (ICS)**. [S.l.: s.n.], 2011. p. 75–85.

INTEL. **Quad-Core Intel ® Xeon ® Processor 5400 Series Datasheet**. [S.l.], 2008. Available from Internet: <http://www.intel.com/assets/PDF/datasheet/318589.pdf>.

INTEL. **Intel® Itanium® Architecture Software Developer's Manual**. [S.l.], 2010.

INTEL. **Intel® Xeon® Processor 7500 Series**. [S.l.], 2010.

INTEL. **2nd Generation Intel Core Processor Family**. [S.l.], 2012. v. 1, n. September.

INTEL. **Intel Performance Counter Monitor - A better way to measure CPU utilization**. 2012. Available from Internet: <http://www.intel.com/software/pcm>.

ITO, S.; GOTO, K.; ONO, K. Automatically optimized core mapping to subdomains of domain decomposition method on multicore parallel environments. **Computers & Fluids**, Elsevier Ltd, v. 80, p. 88–93, jul. 2013.

JEANNOT, E.; MERCIER, G.; TESSIER, F. Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques. **IEEE Transactions on Parallel and Distributed Systems**, v. 25, n. 4, p. 993–1002, abr. 2014.

JEDEC. **DDR3 SDRAM Standard**. 2012.

JIN, H.; FRUMKIN, M.; YAN, J. **The OpenMP implementation of NAS Parallel Benchmarks and Its Performance**. [S.l.], 1999.

JOHNSON, M. et al. PAPI-V: Performance Monitoring for Virtual Machines. In: **International Conference on Parallel Processing Workshops (ICPPW)**. [S.l.: s.n.], 2012. p. 194–199.

KARYPIS, G.; KUMAR, V. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. **SIAM Journal on Scientific Computing**, v. 20, n. 1, p. 359–392, jan. 1998.

KLUG, T. et al. autopin – Automated Optimization of Thread-to-Core Pinning on Multicore Systems. **High Performance Embedded Architectures and Compilers (HiPEAC)**, v. 3, n. 4, p. 219–235, 2008.

LAROWE, R. P.; HOLLIDAY, M. A.; ELLIS, C. S. An Analysis of Dynamic Page Placement on a NUMA Multiprocessor. **ACM SIGMETRICS Performance Evaluation Review**, v. 20, n. 1, p. 23–34, 1992.

LöF, H.; HOLMGREN, S. affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In: **International Conference on Supercomputing (SC)**. [S.l.: s.n.], 2005. p. 387–392.

LUK, C. et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: **ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)**. [S.l.: s.n.], 2005. p. 190–200.

LUSZCZEK, P. et al. **Introduction to the HPC Challenge Benchmark Suite**. [S.l.], 2005.

MAGNUSSON, P. et al. Simics: A Full System Simulation Platform. **IEEE Computer**, IEEE Computer Society, v. 35, n. 2, p. 50–58, 2002.

MARATHE, J.; MUELLER, F. Hardware Profile-guided Automatic Page Placement for ccNUMA Systems. In: **ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)**. [S.l.: s.n.], 2006. p. 90–99.

MARATHE, J.; THAKKAR, V.; MUELLER, F. Feedback-Directed Page Placement for ccNUMA via Hardware-generated Memory Traces. **Journal of Parallel and Distributed Computing (JPDC)**, v. 70, n. 12, p. 1204–1219, 2010.

MARTIN, M. M. K.; HILL, M. D.; SORIN, D. J. Why On-Chip Cache Coherence is Here to Stay. **Communications of the ACM**, v. 55, n. 7, p. 78, jul. 2012.

MARTIN, M. M. K. et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. **ACM SIGARCH Computer Architecture News**, ACM, v. 33, n. 4, p. 92–99, 2005.

MCVOY, L.; STAELIN, C. Lmbench: Portable Tools for Performance Analysis. In: **USENIX Annual Technical Conference (ATC)**. [S.l.: s.n.], 1996. p. 23–38.

MIPS. **MIPS R10000 Microprocessor User's Manual, Version 2.0**. [S.l.: s.n.], 1996.

NETHERCOTE, N.; SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In: **ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)**. [S.l.: s.n.], 2007.

OPENMP. **OpenMP Application Program Interface**. [S.l.], 2013.

PATEL, A. et al. MARSSx86: A Full System Simulator for x86 CPUs. In: **Design Automation Conference 2011 (DAC'11)**. [S.l.: s.n.], 2011.

PELLEGRINI, F. Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In: **Scalable High-Performance Computing Conference (SHPCC)**. [S.l.: s.n.], 1994. p. 486–493.

RADOJKOVIć, P. et al. Thread Assignment of Multithreaded Network Applications in Multicore/Multithreaded Processors. **IEEE Transactions on Parallel and Distributed Systems (TPDS)**, v. 24, n. 12, p. 2513–2525, 2013.

RIBEIRO, C. P. et al. Improving memory affinity of geophysics applications on numa platforms using minas. In: **International Conference on High Performance Computing for Computational Science (VECPAR)**. [S.l.: s.n.], 2011.

RIBEIRO, C. P. et al. Memory Affinity for Hierarchical Shared Memory Multiprocessors. In: **International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. [S.l.: s.n.], 2009. p. 59–66.

SHWARTSMAN, S.; MIHOCKA, D. Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure. In: **International Symposium on Computer Architecture (ISCA)**. Beijing, China: [s.n.], 2008.

SONNEK, J. et al. Starling: Minimizing Communication Overhead in Virtualized Computing Platforms Using Decentralized Affinity-Aware Migration. In: **International Conference on Parallel Processing (ICPP)**. [S.l.: s.n.], 2010. p. 228–237.

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

SWAMY, T.; UBAL, R. Multi2sim 4.2 – a compilation and simulation framework for heterogeneous computing. In: **International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)**. [S.l.: s.n.], 2014.

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: **Workshop on Memory Access on Future Processors: A Solved Problem? (MAW)**. [S.l.: s.n.], 2008. p. 377–384.

THOZIYOOR, S. et al. **Cacti 5.1**. [S.l.], 2008.

TIKIR, M. M.; HOLLINGSWORTH, J. K. Hardware monitors for dynamic page migration. **Journal of Parallel and Distributed Computing (JPDC)**, v. 68, n. 9, p. 1186–1200, sep. 2008.

TOLENTINO, M.; CAMERON, K. The optimist, the pessimist, and the global race to exascale in 20 megawatts. **IEEE Computer**, v. 45, n. 1, 2012.

TORRELLAS, J. Architectures for extreme-scale computing. **IEEE Computer**, v. 42, n. 11, p. 28–35, 2009.

TRAHAY, F. et al. EZTrace: a generic framework for performance analysis. In: **International Symposium on Cluster, Cloud and Grid Computing (CCGrid)**. [S.l.: s.n.], 2011. p. 618–619.

VERGHESE, B. et al. **OS Support for Improving Data Locality on CC-NUMA Compute Servers**. [S.l.], 1996.

VILLAVIEJA, C. et al. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In: **International Conference on Parallel Architectures and Compilation Techniques (PACT)**. [S.l.: s.n.], 2011. p. 340–349.

WANG, W. et al. Performance Analysis of Thread Mappings with a Holistic View of the Hardware Resources. In: **IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)**. [S.l.: s.n.], 2012.

ZHAI, J.; SHENG, T.; HE, J. Efficiently Acquiring Communication Traces for Large-Scale Parallel Applications. **IEEE Transactions on Parallel and Distributed Systems (TPDS)**, v. 22, n. 11, p. 1862–1870, 2011.

ZHOU, X.; CHEN, W.; ZHENG, W. Cache Sharing Management for Performance Fairness in Chip Multiprocessors. In: **International Conference on Parallel Architectures and Compilation Techniques (PACT)**. [S.l.: s.n.], 2009. p. 384–393.