UNIVERSITY OF SANTIAGO DE COMPOSTELA

DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

CENTRO DE INVESTIGACIÓN EN TECNOLOXÍAS DA INFORMACIÓN
(CITIUS)

PhD DISSERTATION

# PERFORMANCE COUNTER-BASED STRATEGIES TO IMPROVE DATA LOCALITY ON MULTIPROCESSOR SYSTEMS: REORDERING AND PAGE MIGRATION TECHNIQUES

Author:
**Juan Ángel Lorenzo del Castillo**

PhD supervisors:
**Prof. Francisco Fernández Rivera**
**Dr. Juan Carlos Pichel Campos**

Santiago de Compostela, October 2011

**Prof. Francisco Fernández Rivera**, professor at the Computer Architecture Group of the University of Santiago de Compostela

**Dr. Juan Carlos Pichel Campos**, researcher at the Computer Architecture Group of the University of Santiago de Compostela

**HEREBY CERTIFY**:

That the dissertation entitled **Performance Counter-based Strategies to Improve Data Locality on Multiprocessor Systems: Reordering and Page Migration Techniques** has been developed by **Juan Ángel Lorenzo del Castillo** under our direction at the Department of Electronics and Computer Science of the University of Santiago de Compostela in fulfillment of the requirements for the Degree of Doctor of Philosophy.

Santiago de Compostela, October 2011

---

**Francisco Fernández Rivera**, Profesor Catedrático de Universidad del Área de Arquitectura de Computadores de la Universidad de Santiago de Compostela

**Juan Carlos Pichel Campos**, Profesor Doctor del Área de Arquitectura de Computadores de la Universidad de Santiago de Compostela

**HACEN CONSTAR**:

Que la memoria titulada **Performance Counter-based Strategies to Improve Data Locality on Multiprocessor Systems: Reordering and Page Migration Techniques** ha sido realizada por **D. Juan Ángel Lorenzo del Castillo** bajo nuestra dirección en el Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela, y constituye la Tesis que presenta para optar al título de Doctor por la Universidad de Santiago de Compostela.

Santiago de Compostela, Octubre de 2011

| **Francisco Fernández Rivera** | **Juan Carlos Pichel Campos** |
| --- | --- |
| Codirector de la tesis | Codirector de la tesis |

**Juan Ángel Lorenzo del Castillo**
Autor de la tesis

***To my parents,***
***Mari Carmen and Juan***

*If you're not spending every waking moment of your life radically rethinking the nature of the world - if you're not plotting every moment boiling the carcass of the old order - then you're wasting your day.*

Douglas Coupland, *JPod*

*The true artist is quite rational as well as imaginative and knows what he is doing; if he does not, his art suffers. The true scientist is quite imaginative as well as rational, and sometimes leaps to solutions where reason can follow only slowly; if he does not, his science suffers.*

Isaac Asimov, *The Roving Mind*

*"So computers are tools of the devil?" thought Newt. He had no problem believing it. Computers had to be the tools of somebody, and all he knew for certain was that it definitely wasn't him.*

Neil Gaiman and Terry Pratchett, *Good Omens*

*Ooh, look at me, I'm an information worker. My job is clean and environmentally friendly and futuristic.*

Douglas Coupland, *JPod*

# Table of Contents

LIST OF FIGURES xi

LIST OF TABLES xv

ACKNOWLEDGEMENTS xvii

ABSTRACT xix

CHAPTER

**1 Introduction** **1**

1.1 The problem in a nutshell . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Thesis statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.3 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.4 Experimental setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
1.5 How this dissertation is structured . . . . . . . . . . . . . . . . . . . . 8

**2 Evaluating the *FinisTerrae* Architecture** **13**

2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.2 FINISTERRAE architecture . . . . . . . . . . . . . . . . . . . . . . . . 14
2.3 Intel Itanium2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.4 Performance Evaluation on FINISTERRAE . . . . . . . . . . . . . . . . . 22
2.5 Performance Evaluation of Dense Codes . . . . . . . . . . . . . . . . . 23
2.6 Performance Evaluation of Sparse Codes . . . . . . . . . . . . . . . . . 29
2.7 Performance model of FINISTERRAE . . . . . . . . . . . . . . . . . . . 37

2.8 Conclusions . . . . . 50

3 **Accessing Hardware Counters on Itanium2 Montvale. The Perfmon interface** **53**
3.1 Introduction . . . . . 53
3.2 Perfmon programming . . . . . 56
3.3 Evaluation of Perfmon on FINISTERRAE . . . . . 60
3.4 Conclusions . . . . . 68

4 **Locality Improvement on Irregular Codes** **71**
4.1 Introduction . . . . . 71
4.2 Locality optimisation technique . . . . . 72
4.3 Locality optimisation using randomly sampled matrices . . . . . 76
4.4 Locality optimisation using hardware counters . . . . . 81
4.5 Locality optimisation using latency information . . . . . 93
4.6 Conclusions . . . . . 101

5 **Page Migration** **103**
5.1 Introduction . . . . . 103
5.2 Development of a page migration infrastructure . . . . . 105
5.3 Operating tests . . . . . 115
5.4 Affinity decisions . . . . . 127
5.5 Access-based migration algorithm for N-cell nodes . . . . . 127
5.6 Latency-based migration algorithm for N-cell nodes . . . . . 129
5.7 Evaluation in a dedicated environment . . . . . 133
5.8 Evaluation in a multiprogrammed environment . . . . . 136
5.9 Conclusions . . . . . 145

6 **Conclusions and Future Work** **147**

**Resumen** **153**

**References** **163**

# LIST OF FIGURES


2.1 HP Integrity rx7640 node (taken from [1]). . . . . . . . . . . . . . . . . . . . . . . . 15
2.2 Block diagram of an HP Integrity rx7640 node. . . . . . . . . . . . . . . . . . . . . 15
2.3 Instruction group boundary. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.4 1.6 Ghz Dual-Core Intel Itanium2 Montvale (9140N) architecture [2]. . . . . . . 18
2.5 Montecito/Montvale Processor Performance Monitor Register Set [2]. . . . . . . 19
2.6 Data Event Address Configuration Register ($PMC_{40}$). . . . . . . . . . . . . . . . 21
2.7 Data Event Address Register Format ($PMD_{32,33,36}$). . . . . . . . . . . . . . . . . 21
2.8 Average L1 DTLB misses per access for the *memspeed* benchmark. . . . . . . . 25
2.9 Latency of memory accesses when the data are allocated in memory local to the core, in memory on the other cell or in the interleaving zone. . . . . . . . . . . . 30
2.10 Compressed-Sparse-Row (CSR) format example and a basic CSR-based sparse matrix-vector (SpMV) product. . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
2.11 SpMV performance using different loop distributions. . . . . . . . . . . . . . . . 34
2.12 Influence of the data allocation on a rx7640 node. . . . . . . . . . . . . . . . . . 35
2.13 Influence of the thread allocation on a rx7640 node. . . . . . . . . . . . . . . . . 36
2.14 Effect of NUMA optimisations on a rx7640 node. . . . . . . . . . . . . . . . . . 38
2.15 Roofs of FINISTERRAE Roofline Model. . . . . . . . . . . . . . . . . . . . . . . . 40
2.16 Ceilings added to the FINISTERRAE Roofline Model. . . . . . . . . . . . . . . . 41
2.17 Locality walls for the SpMV-FINISTERRAE combination. . . . . . . . . . . . . . 42
2.18 Roofline Model of SpMV for matrices `pct20stif` and `exdata`. . . . . . . . . 45
2.19 Experiment #2 setup. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
2.20 Exploiting thread distribution of SpMV (experiment #2) for matrices `pct20stif` (a) and `exdata` (b). . . . . . . . . . . . . . . . . . . . . . . . . . . 49

3.1 Sampling process flowchart . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
3.2 Effect of attaching a context to a monitored thread. . . . . . . . . . . . . . . . . . 59
3.3 The sampling buffer is mapped to user level when it gets full. . . . . . . . . . . . 59
3.4 Set of matrices used to study the SpMV. . . . . . . . . . . . . . . . . . . . . . . . 64
3.5 Ratio between the number of sampled entries of vector $X$ and its size. . . . . . . 66
3.6 Percentage of time consumed by the `Perfmon` and `libpfm` functions on a SpMV program. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

4.1 Calculation example of $n_{\text{elemsw}}(g)$, $a_{\text{elemsw}}(g)$ and $a_{\text{elems}}(g,h)$. . . . . . . . . . . 74
4.2 Example of windows of locality with variable size: sparse matrix example and distance histogram. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
4.3 Examples of `nmos3` randomly sampled matrices. Matrices with 1%, 2%, 5%, 10% and 20% of the nonzeros with respect to the original matrix. . . . . . . . . . 77
4.4 Normalised SpMV performance obtained by the reorderings generated using the locality optimisation technique ($w = 1$) and the information provided by the randomly sampled matrices on the Itanium2 platform. . . . . . . . . . . . . . . . . 79
4.5 Normalised SpMV performance obtained by the reorderings generated using the locality optimisation technique ($w = variable$) and the information provided by the randomly sampled matrices on the Itanium2 platform. . . . . . . . . . . . . . 80
4.6 Example of sampled matrix using the hardware counters. . . . . . . . . . . . . . 83
4.7 Entries per row (red) and sampled percentage (blue) of nine generated matrices. . 87
4.8 `sme3Da` original and sampled matrix generated by the hardware counters. . . . . 88
4.9 Normalised SpMV performance obtained by the reorderings generated using the locality optimisation technique and the information provided by the hardware-counter sampled matrices for windows of size $w = 1$ and variable size. . . . . . . 89
4.10 Overhead reduction of the locality optimisation technique using as a reference the time required to perform the reordering using windows of fixed size $w = 1$ and the original (non-sampled) matrices. . . . . . . . . . . . . . . . . . . . . . . 92
4.11 Figures display, for a particular row size interval ($X$ axis), the percentage of these rows with latency=0 (without sampled nonzeros) in the latency histogram ($Y$ axis). Interval $a$ corresponds to rows of the matrix with $[5a, 5(a+1))$ nonzeros. 95
4.12 Normalised performance of the matrices reordered using latency information with respect to the original reordering technique. . . . . . . . . . . . . . . . . . . . 96
4.13 Average overhead of the reordering technique: comparison with METIS library. . 99

4.14 Normalised performance of the matrices reordered using latency information with respect to the original matrices. . . . . . . . . . . . . . . . . . . . . . . . . . 100

5.1 Time intervals in a profiling process. . . . . . . . . . . . . . . . . . . . . . . . . 106
5.2 Waiting on multiple contexts. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
5.3 Sampling section from our page migration infrastructure. . . . . . . . . . . . . . 109
5.4 Flow diagram from the main function of the monitoring tool. . . . . . . . . . . . 112
5.5 Flow diagram from the interruption handler of the monitoring tool. . . . . . . . . 113
5.6 Number of different cache lines accessed per page and per thread. Only one access per cache line is shown. . . . . . . . . . . . . . . . . . . . . . . . . . . 115
5.7 Total number of cache lines accessed per page and per thread. The sum of all accesses per cache line are shown. Y-axis in log scale. . . . . . . . . . . . . . . . 116
5.8 Sum of latencies of all references to a given page. Y-axis in log scale. . . . . . . 117
5.9 Number of threads accessing a given page. . . . . . . . . . . . . . . . . . . . . . 117
5.10 Access latency from threads on each cell. No migration. . . . . . . . . . . . . . 120
5.11 Access latency from threads on each cell. Migration. . . . . . . . . . . . . . . . 120
5.12 Migration throughput (MB/s) between Cells 0 and 1. . . . . . . . . . . . . . . . 123
5.13 Average distance for 2 and 4 threads . . . . . . . . . . . . . . . . . . . . . . . . 124
5.14 Average distance for 8 and 16 threads . . . . . . . . . . . . . . . . . . . . . . . 125
5.15 HP Superdome node . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128
5.16 Superdome Node Graph . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128
5.17 Number of accesses and latencies for BT. No page migrations. . . . . . . . . . . 139
5.18 Number of accesses and latencies for BT. Access-based page migration. . . . . . 140
5.19 Number of accesses and latencies for BT. Latency-based ($Lat_{real}$) page migration. 141
5.20 Number of accesses and latencies for BT. Latency-based ($Lat_{avg}$) page migration. 142

# LIST OF TABLES


1.1 Matrix test set used in this dissertation. . . . . . . . . . . . . . . . . . . . . . . . 9

2.1 Median memory access latency (in ticks) of *memtest* benchmark for different configurations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.2 Duration (ticks/access) of the *producer-consumer* benchmark to transfer the allocated data between two cores. . . . . . . . . . . . . . . . . . . . . . . . . . 28
2.3 Matrix benchmark suite used in the performance evaluation. . . . . . . . . . . . 32
2.4 Imbalance of matrices `pct20stif` and `exdata_1`. . . . . . . . . . . . . . . . 44

3.1 List of the matrix test set and percentage of sampled entries by `Perfmon` for 2, 4, 8 and 16 threads. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

4.1 Matrix benchmark suite used in our sampling tests. . . . . . . . . . . . . . . . . . 76
4.2 Characteristics of the sampled matrices generated by the hardware counters. . . . 88
4.3 SpMV performance comparison between the original matrices and the reorderings obtained by the locality optimisation technique (in GFLOPS). . . . . 91
4.4 Comparison between the latency histograms of SpMV iterations 2, 4 and 6. . . . 97
4.5 Number of windows of locality using different criteria in the windows creation process. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99

5.1 Average execution time (sec.) of a data-intensive benchmark, with and without page migration. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
5.2 Performance of the OpenMP NAS suite using the default first-touch policy and the page migration strategies proposed in a dedicated environment. . . . . . . . . 135

5.3 Performance of the OpenMP NAS suite using the default first-touch policy and the page migration strategies proposed in a multiprogrammed environment. . . . 138
5.4 Locality statistics of BT for each page migration strategy. . . . . . . . . . . . . . 145

# Acknowledgements

What a long, rewarding and enthralling journey! Although research is sometimes portrayed as a lonely job, shut away in a laboratory far from the madding crowd, the truth is that it is actually a team work. During these years, I have come across a lot of people who have influenced me one way or another in my personal and professional life. None of this work would have been accomplished if it had not been for their advice and support. Now is the time to share the credit and show my gratitude to them all.

First and foremost, I would like to thank Francisco F. Rivera (Fran) and Juan Pichel, my PhD advisors, for their guidance towards the completion of this thesis. The synergy of mature experience and witty youth has turned out to be the perfect recipe to achieve fruitful results.

To Tomás F. Pena, Jose Carlos Cabaleiro (Caba) and Dora Blanco. I have learnt a lot from them during our discussions and project meetings. Their suggestions and help proved invaluable. To me, they are also advisors of this thesis. I would also like to thank David E. Singh, for the technical discussions and ideas that have positively impacted my work.

To the Department of Electronics and Computer Science. I could not have found a more enriching environment to develop my work. Their members are a truly model of professionalism and good rapport, and their parties are epic!

To my colleagues and friends from my department who have accompanied me during these years: María, Ronald, Óscar, Cris, Xulio, Julián, Bea, Fabián, Josito, Jose Carlos, Noe, Álvaro, Sergio, Diego, Manolo and Adri. They have been there for me in the good and bad times, and are now part of my life. I want to thank them all and also the rest of folks at the department for the shared moments and the countless discussions about the most outlandish ideas.

To Montserrat Bóo. Thanks for the conversations, the support, the advice and the sincere friendship.

To Petr Tůma, member of my PhD committee and host during my two research stays at the Department of Distributed and Dependable Systems of Charles University in Prague. He and his group have been a reference of good methodology and well-done work. I did not enjoy the food, though.

To Mark Bull, Gavin J. Pringle and Catherine Inglis at EPCC in Edinburgh. In addition to the academic benefits, my research stay in UK was an unforgettable experience from a personal perspective.

To Stéphane Eranian and the `Perfmon` mailing list. The knowledge gained from them has been a major input to my thesis.

To my present and former teachers from my graduate and undergraduate courses, in Santiago and Valladolid. They are largely responsible for who I am today and my vocation for research.

To my friends from Galicia and Valladolid: Sonia, Susana and Pablo, Luismi and Marta, Álvaro, Nuria, Sara and Eva. So many years and neither time nor distance has diminished our friendship.

I want to express my appreciation to the institutions without whose funding this work would not have been possible. The development of this dissertation was partially supported by Hewlett-Packard under contract 2008/CE377, by the Spain's Ministerio de Educación y Ciencia through the research grant TIN2007-67537-C03-01, FEDER funds under contract TIN2010-17541, Spain's Xunta de Galicia under contract 2010/28 and project 09TIC002CT, and Spain's High Performance Computation in Heterogeneous Architectures Network (CAPAP-H) through research grant TIN2007-29664-E. I also wish to thank the supercomputer facilities provided by the Galicia Centre of Supercomputing (CESGA) and the High Performance and Embedded Architecture and Compilation European Network (HiPEAC-2). Early work on this dissertation was funded by Spain's Xunta de Galicia *María Barbeito* predoctoral contract.

Por último, y no por ello menos importante, quiero dar las gracias a mi familia. A mis padres, Mari Carmen y Juan, por su apoyo y fe en mi incluso cuando las fuerzas flaqueaban. A mi hermana Ana María, a Mariano y a mis sobrinillos, Sandra y Gonzalo. Gracias a todos por estar ahí a lo largo de este viaje.

Santiago de Compostela, October 2011

# Abstract

Over the last years, we have witnessed an important evolution in the available computational resources in science and engineering. The line that has traditionally separated multicomputers from multiprocessors is getting blurred, and nowadays most modern supercomputers include several multicore, NUMA (*Non Uniform Memory Access*) multiprocessor nodes interconnected by a high-speed network. In this context, data locality becomes a subject of paramount importance for the performance of parallel codes.

As systems have grown in complexity, the need for understanding what is happening inside a program has also increased. *Profiling*, understood as a performance monitoring technique that records information about a running code, has proven very useful to narrow down its bottlenecks. In this way, the *performance monitoring hardware counters*, included in the vast majority of modern microprocessors, provide an essential tool to monitor and gain an insight into the system during the execution of a program.

Recently, a new player came on stage. *Precise Event-Based Sampling* (PEBS) is a performance counter-based profiling technique that has been enhanced in the Intel Itanium family with respect to their predecessors. Their performance counters have reached a precision level to the point of returning not only the exact address at which an event occurs, but also the latency of that access. This opens the door to the development of new performance techniques based on that information.

In this dissertation, we approach the study of PEBS techniques to improve the performance of applications on a NUMA, Itanium2-based system. We demonstrate that a low-cost, PEBS profiling can support strategies to improve the performance of an important group of computational and scientific codes in runtime. In addition, the accurate information provided by the new *Event Adress Registers* (EAR) of the Itanium2 architecture helps foster the development of new data allocation strategies. Following this line, we have also developed

a series of dynamic page migration PEBS strategies. Specifically, two problems are addressed: how to improve the performance of locality optimisation techniques for irregular codes in runtime, particularising for the *Sparse Matrix-Vector product* kernel, and how to develop strategies for dynamic page migration.

The main contributions of this dissertation are:

1. A study of the different factors that affect the performance, as well as data and thread allocation policies, in the FINISTERRAE supercomputer, the target platform in which this thesis relies on.
2. The implementation of a performance model for FINISTERRAE.
3. The development of hardware counter-based strategies to assist reordering techniques for irregular codes in order to reduce their cost and improve their behaviour.
4. The development of novel hardware counter-guided, dynamic page migration algorithms that take advantage of the new features provided by the PEBS.

As a software contribution, we present a user-level page-migration framework to monitor, sample and control an application in runtime.

# 1

# *Introduction*

Understanding the performance of a program requires answering the question: *how and where is the time spent?*. Factors such as the underlying system or the type of workload can lead to *bottlenecks*, or points where most of the time is spent. These points can be identified by the action of collecting information related to how an application or system performs when executing, known as *performance monitoring*. Characterising the nature and cause of the bottlenecks using this information allows us to answer our initial question, understanding therefore why a program behaves in a particular way.

Performance monitoring is particularly important in modern shared-memory, multiprocessor systems. The interplay of the cache coherency and consistency, memory hierarchy, buses and processors is fairly complicated and far from being intuitive. Hence, these systems cannot be accurately modelled, so other techniques must be used to characterise the behaviour of an application executed in such systems. *Profiling* is a time-based sampling technique that records information at regular intervals and is based on the statistical fact that the more often an address is issued, the more likely is that the program spends more time there [3].

While the classic application of profiling has been to collect data in order to find out what is going wrong, another not-so-widespread use consists in applying that information to take effective decisions either in runtime or in a pre-execution stage – regarding, among others, data locality or load balancing – that may lead to a performance improvement.

A research field where the data locality is of paramount importance is the case of irregular codes. An application is said to be irregular (as opposite to a regular one) if it presents indirections that prevents us from finding out, at compile time, the set of memory accesses performed by the application. Examples of these accesses are those performed by pointers, the indirection arrays whose content is unknown at compile time, or the use of external functions that access memory and whose structure cannot be determined a priori. Irregular codes present

a low locality and, due to the unpredictability of their accesses, the effective reuse of the memory hierarchy is scarce. Hence, the memory hierarchy is the most important bottleneck for the efficient execution of most of these codes and, therefore, one of the issues where performance can be improved best. Since irregular codes are the core of many important scientific applications, several widespread techniques to improve their data locality exist in the literature [4, 5, 6, 7, 8], many of them developed for *Symmetric Multi-Processor* systems (SMP). Among them, those based on data reordering techniques [9], whose major drawback is the cost of the analysis stage.

Nowadays, performance monitoring counters, also known as *hardware counters*, are a powerful monitoring mechanism included in the PMU (*Performance Monitoring Unit*) [3] of most of the modern microprocessors. Their use is gaining popularity as an analysis and validation tool for profiling, since their effect in the monitored program is virtually imperceptible and their precision has noticeably increased thanks to the new *Precise Event-Based Sampling* (PEBS) [10] features. However, although the PMU can harvest very useful information, it is not always exploited to the fullest of its capabilities. Indeed, the lack of standard tools and libraries to program the hardware counters and access all the information that can be collected keep them restricted to very specific issues, so that many of the possibilities that they offer have not been fully exploited.

In this dissertation, I address the problem of using hardware counters in situations in which they can support strategies to improve the performance of an important group of applications in runtime. In particular, shared-memory parallel codes are considered in the Intel Itanium architecture. This chapter provides an introduction to the work presented in this dissertation, including the structure and contributions of each chapter.

## 1.1 The problem in a nutshell

The problem I address in my dissertation can be flesh out in two specific issues:

- How to improve the performance of locality optimisation techniques for irregular codes in runtime. Particularly, the *Sparse Matrix-Vector product* (SpMV) irregular kernel.
- How to develop strategies for page management to improve the data locality of parallel codes in runtime.

The approaches to both issues share a common denominator: they take advantage of the PEBS features introduced by the hardware counters in the Intel Itanium architecture to provide low-cost, meaningful information that can be helpful in the decision-taking process.

Generally, many of the strategies to improve data locality on irregular codes involve a preliminary stage in which the data are inspected and reordered attending to memory locality criteria. Typically, the overhead of this stage is important and increases with the size of the data. In this dissertation, I demonstrate how some of these reordering-based strategies can benefit from an important overhead reduction and improve their performance by using incomplete information from a hardware counter-based profile.

Regarding page management, most of the parallel applications have not been programmed taking into account that their data will be allocated in the memory of a NUMA (*Non-Uniform Memory Access*) underlying architecture. The cost of accessing data that is physically allocated in a remote memory can unnecessarily increase the execution time of an application. I will demonstrate how a user-level application can profile a target program and, based on algorithms that rely on the hardware support of performance counters, smartly migrate data in runtime to reduce the execution time.

## 1.2 Thesis statement

The main objective of this dissertation is to effectively use on-chip hardware counters to improve the performance of the memory accesses in runtime on the Itanium2 architecture. Two different contexts have been considered: the efficient execution of irregular codes and the development of page migration strategies to improve the execution of parallel codes.

To accomplish these objectives, the following goals must be met:

G1-ARCHEVAL (*Architecture Evaluation*): The first step consists of acquiring a good understanding of the architecture in which this dissertation frames. Issues such as the memory and thread allocation policies, cache coherency and consistency, the NUMA factor, the bus latency and bandwidth and others that may affect the performance need an experimental assessment.

G2-HWCSTUDY (*Hardware Counters Study*): The use of hardware counters as a source of information to guide runtime optimisations of sequential and parallel irregular codes must be validated. This entails establishing procedures to get, store and manage the

information they provide. The existence of specific libraries to access the newest features of the Itanium2 PMUs is scarce, as well as their documentation. As a part of this validation process, the available libraries are analysed to select one that meets our requirements.

G3-REORDTECHIMPRV (*Reordering Techniques Improvement*): This goal aims to characterise locality (between pairs of different references) and affinity (between references and processes) of irregular accesses using models based on the information provided by the hardware counters. The existing reordering techniques for irregular codes can benefit from these models to improve their performance.

G4-PAGEMIGINFR (*Page Migration Infrastructure*): Prior to developing strategies for page migration, a software infrastructure to support the implementation and test of such strategies is required. It must provide a low-cost monitoring system and a mechanism to migrate pages in runtime when required.

G5-PAGEMIGALG (*Page Migration Algorithms*): Comprises the development of hardware counter-based solutions to improve the execution of parallel codes in shared memory multiprocessors using page migration.

## 1.3 Related Work

Many works that deal with the optimisation of the sparse matrix-vector product can be found in literature. Techniques for increasing the locality of SpMV can be mainly divided into two groups: data reordering and code restructuring techniques.

Standard reordering techniques are considered classical methods for increasing the locality in the execution of the SpMV code. The most used techniques are the bandwidth reduction algorithms, which derive from the Cuthill-McKee algorithm [11]. In a recent work [12], authors evaluate *minimum degree*-based heuristics such as the *approximate minimum degree* algorithm [13] on multicore processors. Oliker *et al.* [14] show the benefits that are offered by the application of some of these reordering algorithms to sparse codes when executed on different multiprocessor architectures. Note that, unlike standard reordering algorithms, the main objective of the data reordering technique considered in this work is to increase the data reuse and, as a consequence, the data locality. Coutinho *et al.* [15] perform a comparison of

different data reordering algorithms for the SpMV in edge-based unstructured grid computations. However, they only focus on serial executions.

Techniques based on restructuring the code, like *blocking* or *tiling*, have been successfully applied to different irregular codes such as the product of a sparse matrix by a dense matrix [16, 17] and stationary iterative methods [18]. Im *et al.* [19] propose register and cache blocking as optimisation techniques for the SpMV. In [20], a performance model for the blocked SpMV is presented, which allows picking in nearly all cases the actual optimal blocksize. In these last two works the authors use a randomly sampled matrix at runtime to detect the best blocking size. Vuduc *et al.* [21] extend the notion of blocking in order to exploit variable block shapes by decomposing the original matrix to a proper sum of submatrices storing each submatrix in a variation of the blocked CSR format. In a recent work [22], a comparative study of different blocking storage techniques for sparse matrices on several multicore platforms is performed. One of the main drawbacks of these techniques is the strong dependence with the sparsity pattern of the matrix. For example, register blocking only achieves good performance for matrices with small dense-blocks in the pattern. Unlike these solutions, our locality optimisation technique is effective for matrices with any kind of pattern [12]. Finally, Belgin *et al.* [23] introduce a representation for sparse matrices based on the observation that many matrices can be divided into blocks that share a small number of different patterns. The goal is to reduce the SpMV memory bandwidth requirements by reducing the index overhead.

Some authors have demonstrated that both groups of techniques are complementary. In particular, Toledo [24] evaluates different standard reordering techniques and combines them with blocking, showing that SpMV performance increases significantly depending on the size and sparseness of the considered matrix. Pinar and Heath [25] introduce a reordering technique that favors the creation of dense blocks on the pattern of the sparse matrix, and in this way the efficiency of the blocking technique proposed by Toledo is increased. Moreover, a comparison between their reordering technique and some standard reordering techniques is carried out. In another work [26], a combination of data reordering algorithms and register blocking has been applied to the SpMV on shared memory multiprocessors, finding little benefit. The locality optimisation technique used in the present dissertation can also be applied to codes where data are stored using a blocked scheme. An example was published in [27] where a reordering of the sparse matrix in combination with blocking techniques was successfully applied to the SpMV. The technique was evaluated on different uniprocessors and on distributed memory multiprocessors.

Moreover, there are several papers that deal with the SpMV optimisation problem using compression. In a recent work [28], the authors propose two different compression methods targeting index and numerical values. Williams *et al.* [29] apply an index reduction technique, in which 16-byte indices are used when it is possible. In the same work the authors propose several additional optimisation techniques for the SpMV, which are evaluated on different multicore platforms. Authors examine among others: software pipelining, prefetching approaches, register and cache blocking, etc. Nevertheless, they do not consider data reordering techniques in order to increase locality.

Research regarding the use of hardware counters is mainly focused on the characterisation and analysis of possible bottlenecks in the performance of applications [30, 31]. However, some works use hardware counters for different optimisations such as improving cache utilisation [32], reducing memory access stalls [33], selecting compiler optimisation settings [34] and dynamic page migration [35]. To the best of our knowledge, researchers have never dealt with the locality optimisation of the SpMV in runtime using only the information provided by the hardware counters.

Regarding strategies for page management, numerous works study the proper memory placement of data on NUMA systems. These studies have usually been performed in UNIX systems such as Solaris or Irix. In [36], Antony *et al.* compare the NUMA awareness of Solaris and Linux, stating the fact that the NUMA support in Linux is more recent (from kernel 2.6) than in Solaris and, therefore, lacks some features that have been present for a long time in other systems. As an example, some modes of memory placement, such as *next-touch* policies, are not available yet in the Linux mainstream kernel.

Most of the developed migration techniques can be categorised in those developed as a user-level tool and those somehow embedded in the operating system kernel. Tikir *et al.* [35] introduce a user-level, profile-driven page migration scheme using performance counters on an Sun Fire 6800 server. Their migration algorithm is based on the number of times a page is accessed by a processor, and uses the built-in *move-on-next-touch* feature of Solaris 9. It achieves a time improvement in some applications by up to 16%, although it requires inserting instrumentation code into the monitored application. The same authors had also previously proposed a dynamic user-level page migration scheme based on an approximate trace of memory accesses obtained by sampling the network interconnect [37]. Marathe *et al.* [38] introduce a hardware-assisted page placement scheme based on automated profiling on a SGI Altix architecture. They use the *libpfm* library to access the hardware counters of the Itanium2 processors. Their method firstly runs a truncated version of an OpenMP program to extract an

approximate trace of its memory accesses. Then, the program is effectively executed, forcing the page placement by touching its pages so that the first-touch feature places them in the desired allocation. Therefore, once the data have been placed, they cannot be reallocated. The authors admit that their solution involves a substantial execution overhead that erases the gain for several benchmarks. Closely related to the previous work, Thakkar [39] uses *libpfm* on Itanium2-based, SGI Altix and x86-64 Opteron platforms and studies the use of hardware counters to assist dynamic page placement on Linux. His proposal of a latency-based algorithm considers that the access latencies from a given node are constant. His work on the Itanium2 platform was abandoned due to the instability of the traces obtained, and his results on Opteron show an improvement of 8-15%.

Bull and Johnson study the tradeoffs between page migration, replication and data distribution for OpenMP applications on the Sun WildFire system [40]. They suggest that page replication can be even more beneficial than migration.

Tao *et al.* propose three page migration algorithms supported by memory access histograms on a *shared memory in a Lan-like Environment* PC clusters [41]. Their algorithms require a large amount of references issued before a migration decision can be taken. Additionally, they assume that if a page is accessed, the neighbouring pages will be also accessed by the same node.

Nikolopoulos *et al.* present in [42] and [43] two algorithms for moving virtual memory pages to the nodes that reference them more frequently on an IRIX 6.5.5 system. The first one, for OpenMP iterative codes, assumes that the page reference pattern of one iteration will be repeated throughout the execution of the program. The second algorithm checks for hot memory areas and migrates the pages with excessive remote references.

A more recent work of Nikolopoulos *et al.* dynamically collocates threads and memory affinity sets of iterative programs in the presence of unpredictable scheduler interventions [44]. The architecture tested was a SGI Origin2000. It takes into account the impact of thread migrations and preemptions as well as the memory placement based on a speculative page migration criterion. This proposal requires compiler support by linking the monitored program to a page-migration library.

Wilson and Aglietti [45] implement a Dynamic Page Placement (DPP) strategy by a replication/migration decision tree on a cc-NUMA multiprocessor simulator, proposing several ideas for improving DPP.

One of the most renowned approaches to kernel-level dynamic page migration is presented by Goglin *et al.* in [46] and [47]. The authors develop an implementation of a next-touch

memory placement for the Linux kernel in the i386 architecture given that, contrary to other systems such as Solaris, Linux lacks one. Their proposal modifies the kernel to have a page migrated close to the thread that last accessed it. This implementation is yet to be included in the mainstream kernel.

## 1.4 Experimental setup

The work outlined in this dissertation involves testing locality improvement techniques. The *Sparse Matrix-Vector Product* (SpMV) has been used as a test irregular kernel. A wide set of sparse matrices with different features has been chosen as an input for this kernel.

There exist several sparse matrix compilations available. The most renowned are the *Matrix Market collection* [48] and the *Tim Davis collection* from the University of Florida (UFL) [49]. These compilations provide a large number of matrices from numerous applications which range from finite element problems (FEM) to device simulation problems.

Table 1.1 enumerates the set of square matrices used throughout this dissertation. Their dimensions (*N*), number of nonzeros (*NNZ*), ratio of nonzero elements per row, application field and chapter in which each of them have been used are shown.

## 1.5 How this dissertation is structured

The work developed in this thesis unfolds in the following chapters:

**Chapter 2: Evaluating the FinisTerrae Architecture.** This chapter describes the architecture of the supercomputer in which this dissertation has been carried out. The performance of dense and sparse codes is evaluated in order to define some thread and data allocation recommendations and, finally, a *Roofline Model* is implemented for a node of this system. This chapter expands on the work presented in the following papers:

Juan A. Lorenzo, Francisco F. Rivera, Dora B. Heras, José C. Cabaleiro, Tomás F. Pena, Juan C. Pichel and David E. Singh. *Thread Allocation Issues for Irregular Codes in the Finisterrae System*. XX Jornadas de Paralelismo. A Coruña, Galicia, Spain, September 2009.

Juan A. Lorenzo, Petr Tůma, Juan C. Pichel, and Francisco F. Rivera. *On the Influence of Thread Allocation for Irregular Codes in NUMA Systems*. 10th Int. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT). Hiroshima, Japan, December 2009.

| Matrix | $N$ | $NNZ$ | $\frac{NNZ}{N}$ | Description | Chapter(s) |
|---|---|---|---|---|---|
| af23560 | 23560 | 484256 | 21 | Airfoil eigenvalue calculation | 3 |
| av41092 | 41000 | 1683902 | 41 | Irregular finite-element | 2 |
| bcsstk04 | 132 | 1890 | 14 | Symm. stiffness matrix, oil rig | 2 |
| bcsstk05 | 153 | 1288 | 8 | Transmission tower | 2 |
| bcsstk17 | 10974 | 219812 | 20 | Elevated pressure vessel | 3 |
| bcsstk18 | 11948 | 80519 | 7 | R.E. Ginna Nuclear Power Station | 3 |
| bcsstk20 | 485 | 1810 | 4 | Frame within a suspension bridge | 3 |
| bcsstk23 | 3134 | 24156 | 8 | Part of a 3D globally triang. building | 3 |
| bcsstk24 | 3562 | 81736 | 23 | Winter sports arena | 3 |
| bcsstk25 | 15439 | 133840 | 9 | Columbia 76-story skyscraper | 3 |
| bcsstk28 | 4410 | 111717 | 25 | Solid element model, linear statics | 3 |
| bfw398b | 398 | 2910 | 7 | Bounded Finline Dielectric Waveguide | 3 |
| crystk03 | 24696 | 1751178 | 71 | FEM crystal free vibration matrix | 4 |
| e40r0000 | 17281 | 553956 | 32 | Driven cavity | 3 |
| e40r0100 | 17000 | 553562 | 33 | Fluid dynamics | 2 |
| exdata_1 | 6000 | 2269501 | 378 | Linear equations | 2 |
| fidap019 | 12005 | 259863 | 22 | Finite element modeling | 3 |
| fidapm29 | 13668 | 186294 | 14 | Finite element modeling | 3 |
| garon2 | 13535 | 390607 | 29 | 2D FEM, Navier-Stokes | 2, 4 |
| gyro_k | 17361 | 1021159 | 59 | Bone Micro-Finite Element | 2, 4 |
| lnsp131 | 131 | 536 | 4 | Fluid flow modeling | 3 |
| lnsp3937 | 3937 | 25407 | 6 | Fluid flow modeling | 3 |
| mbeaflw | 496 | 49920 | 101 | Economic modeling | 3 |
| memplus | 17758 | 126150 | 7 | Electronic circuit design | 3 |
| mhd416a | 416 | 8562 | 21 | Magnetohydrodynamics | 3 |
| mhd4800b | 4800 | 27520 | 6 | Magnetohydrodynamics | 3 |
| mixtank_new | 29957 | 1995041 | 67 | POLYFLOW mixing tank | 2, 4 |
| msc10848 | 10848 | 1229778 | 113 | Structural engineering | 2, 4 |
| nd3k | 9000 | 3279690 | 364 | ND problem | 2, 4 |
| nmos3 | 18588 | 386594 | 21 | Semiconductor device simulation | 2, 4 |
| pct20stif | 52329 | 2698463 | 52 | CT20 Engine Block - Stiffness matrix | 2, 4 |
| psmigr_1 | 3140 | 543162 | 173 | Intercounty migration | 2, 3 |
| rajat15 | 33000 | 443573 | 13 | Circuit simulation | 2 |
| sherman2 | 1080 | 23094 | 21 | Oil reservoir simulation | 3 |
| sme3Da | 12504 | 874887 | 70 | 3D structural mechanics problem | 2, 4 |
| syn12000a | 12000 | 1436806 | 120 | Linear equations | 2 |
| tsyl201 | 20685 | 2454957 | 119 | Part of condeep cylinder | 2, 4 |
| west0381 | 381 | 2157 | 6 | Chemical Engineering plant | 3 |
| west2021 | 2021 | 7353 | 4 | Chemical engineering plant | 3 |

**Table 1.1:** Matrix test set used in this dissertation.

Juan A. Lorenzo, Juan C. Pichel, David LaFrance-Linden, Francisco F. Rivera and David E. Singh. *Lessons Learnt Porting Parallelisation Techniques for Irregular Codes to NUMA Systems.* 18th Euromicro Conference on Parallel, Distributed and Network based Processing (PDP). Pisa, Italia, February 2010.

Juan A. Lorenzo, Juan C. Pichel, Tomás F. Pena, Marcos Suárez and Francisco F. Rivera. *Study of Performance Issues on a SMP-NUMA System using the Roofline Model*. 17th Int'l Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11). July 18-21, 2011, Las Vegas, USA.

Oscar G. Lorenzo, Juan A. Lorenzo, J.C. Cabaleiro, Dora B. Heras, Marcos Suárez and Juan C. Pichel, *A Study of Memory Access Patterns in Irregular Parallel Codes Using Hardware Counters Based Tools*. 17th Int'l Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11). July 18-21, 2011, Las Vegas, USA.

**Chapter 3: Accessing Hardware Counters on Itanium2 Montvale. The `Perfmon` interface.** The adoption of an interface that provides an adequate access to the hardware counter features is studied. The `Perfmon` interface is tested and analysed in the target platform, evaluating its access to the features provided by the PEBS in several scenarios. The papers pointed out in Chapter 2 can also be included as contributions for this chapter.

**Chapter 4: Locality Improvement on Irregular Codes.** This chapter assesses the feasibility of increasing the locality of irregular codes –particularly for the sparse matrix-vector product– in which only a subset of the memory accesses, that are provided by the PEBS, is available in the optimisation process. The work presented in this chapter expands on the work developed in the following papers:

Juan C. Pichel, Juan A. Lorenzo, Dora B. Heras and José C. Cabaleiro. *Evaluating Sparse Matrix-Vector Product on the FinisTerrae Supercomputer*. 9th Int. Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE). Gijón, Spain, June 2009.

Juan C. Pichel, Juan A. Lorenzo, Dora B. Heras, José C. Cabaleiro and Tomás F. Pena. *Analyzing the Execution of Sparse Matrix-Vector Product on the Finisterrae SMP-NUMA System.* Journal of Supercomputing, pages 1-11, Springer Netherlands, ISSN 0920-8542, 2010.

Juan C. Pichel, Juan A. Lorenzo, Francisco F. Rivera, Dora B. Heras and Tomás F. Pena. *Using sampled information, is it enough for the SpMV locality optimization?*. Concurrency and Computation: Practice and Experience, 2011 (*under review*).

**Chapter 5: Page Migration.** As a software contribution, an infrastructure to dynamically monitor an application and perform data migration is presented. This chapter relies on this infrastructure to evaluate novel PEBS-based migration algorithms.

This dissertation concludes with Chapter 6, discussing the results achieved and the future work that may follow from this work.

# 2

# *Evaluating the FinisTerrae Architecture*

## 2.1 Introduction

As mentioned in Chapter 1, most of the existing irregular kernels were conceived to work in SMP systems, where the memory access latency is the same for all processors. However, state-of-the-art architectures involve many cache levels in complex several-node NUMA configurations with different number of multi-core processors. A good example is the supercomputer FINISTERRAE installed at the *Galicia Supercomputing Centre* (CESGA) [50] in Spain. FINISTERRAE is a SMP-NUMA system with more than 2500 processors, 19 TB of RAM memory, 390 TB of disk storage and a 20Gbps Infiniband network, orchestrated by a SuSE Linux distribution. Designed to undertake great technological and scientific computational challenges, it is one of the largest shared-memory supercomputers in Europe.

In such a complex infrastructure, the observations we can make are not straightforward, because the interplay of cache contention, bus contention, cache coherency and other mechanisms is far from transparent and therefore requires experimental assessment. In this context, the memory allocation and the thread-to-core distribution may become very important in the performance of a generic code and, more noticeably, in irregular codes, whose structure does not make an efficient use of the cache hierarchy, as it happens in iterative kernels (*sparse matrix-vector multiplication*, *irregular reduction*, etc). Hence, different latencies, depending on the processor the data are assigned to, can significantly affect the performance.

Since the work presented in this dissertation has been carried out in this supercomputer, the first step consisted in acquiring a good understanding of its architecture. This is a process that cannot rely on browsing through manuals and datasheets solely, but it requires to undertake a hands-on assessment of how both dense and irregular techniques behave on this machine. This chapter meets the goal G1-ARCHEVAL stated in Section 1.2.

The structure of this chapter is as follows: the following section outlines the FINISTERRAE infrastructure focusing on the architecture of a single node. Next, a brief review of its Itanium2 Montvale processors is presented, subsequently exploring in more detail its Performance Monitoring Unit. An in-depth explanation of some of their performance counters is also carried out. Once presented the architecture, the actual evaluation of the system is undertaken. Several dense and sparse tests were used to study the influence and impact on the performance of thread and memory allocations. Moreover, a well-known insightful performance model was adapted to FINISTERRAE to graphically illustrate our observations. Finally, some discussion and conclusions are drawn.

## 2.2 FINISTERRAE architecture

FINISTERRAE is an SMP-NUMA machine which comprises 142 HP Integrity rx7640 computation nodes. Each node consists of eight 1.6Ghz-DualCore Intel Itanium2 Montvale (9140N) processors arranged in a two-SMP-cell NUMA configuration[1]. Figure 2.1 shows a diagram of a cell related to the I/O system and to the other cell. Figure 2.2 shows the block diagram of a whole node as well as its core disposition. As seen in the former, each cell is composed of two buses at 6.8 GB/s, each connecting two sockets (four cores) to a 64GB memory through a *sx2000* cell controller. This controller maintains a directory-based, cache-coherent memory system and connects both cells through a 27.3 GB/s crossbar. It yields a theoretical peak processor bandwidth of 13.6 GB/s and a peak memory bandwidth of 16 GB/s (four buses at 4 GB/s).

The main memory address range handled by the cell controller can be split in two modes: three fourths of the address range map to addresses in the local memory, the remaining one fourth maps in an interleaved manner to addresses in both local and remote memory.

## 2.3 Intel Itanium2

The Intel Itanium architecture (formerly called IA-64) broke with the past when it first appeared in 2001. Developed together with Hewlett-Packard, it implements an *Explicitly Parallel Instruction Computing* (EPIC) instruction set, to the detriment of other traditional ones such as CISC and RISC [51]. EPIC [52] was conceived with the goal of moving beyond RISC performance bounds with explicitly-parallel instruction streams. Typically, traditional

[1] There exists a 143th Superdome node composed of 128 Montvale cores and 1 TB memory.

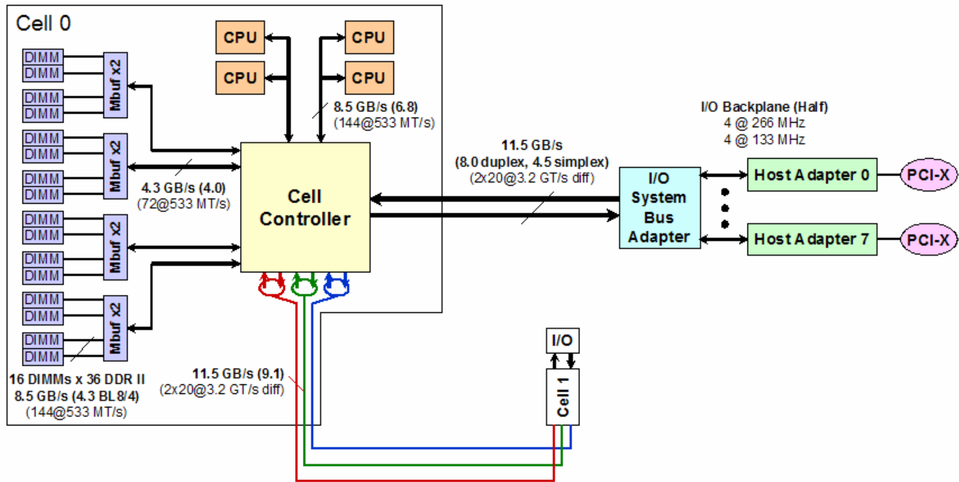



**Figure 2.1:** HP Integrity rx7640 node (taken from [1]).

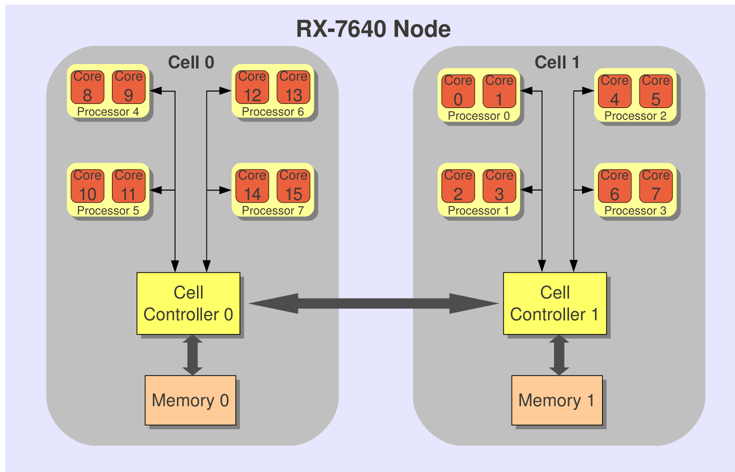



**Figure 2.2:** Block diagram of an HP Integrity rx7640 node. Four dual-core Itanium2 Montvale are connected to their local memory in every cell through a Cell Controller, which uses a crossbar to communicate with an identical second cell.

```
mov r31 = ip //1st group          { .mii
ld r2 = [r3];; //1st group           mov r31 = ip
st [r4] = r2 //2nd group          }
add r5 = r6, r7 //2nd group       { .mmi
                                     ld4 r2 = [r3];; //load
                                     st4 [r4] = r2 //store
                                     add r5 = r6, r7 //int op
                                  }
```

**Figure 2.3:** Instruction group boundary.

architectures showed a limited parallelism where a compiler translated a source code into a sequential machine code. This machine code was interpreted by the hardware in order to extract the maximum parallelism in run-time, so that most of the multiple CPU functional units were efficiently used. Unfortunately, this is a hard task to perform, and traditional processors are often about 60% idle.

Whereas RISC architectures approach a processing limit of one instruction per clock cycle, the Explicit Parallelism allows multiple instructions in one cycle. The parallelisation effort is now delegated to the compiler, which is tightly tied to the target architecture and whereby it owns a wider scope of the whole execution scenario. This fact enables the compiler to determine in advance which instructions can be executed in parallel, generating a parallel machine code which allows a more efficient use of the execution resources.

Itanium2 implements a form of Very Long Instruction Word (VLIW). Each instruction word contains three 41-bit instructions together with a 5-bit template field grouped into 128-bit aligned containers called "bundles", which can only be treated as atomic [53]. Each instruction group is selected by the compiler to present no RAW or WAW dependencies, so that instructions can be issued in parallel depending on the available resources. Whenever possible, the compiler can take advantage of this, allowing the processor to execute up to 2 bundles (6 instructions) per cycle, which is claimed to be one of the main advantages of EPIC paradigm. Figure 2.3 shows an example of a sequence of instructions (left) and a bundle (right). On the left, the instructions have been splitted up in two groups (delimited by two semi-colons) to avoid the RAW dependency caused by the accesses to `r2`. Note that a single instruction group can span multiple bundles, which is what is actually done in this case: the assembler can only package instructions into some predefined bundle formats, such as `.mii` (*memory-instruction-instruction*) or `.mmi` (*memory-memory-instruction*). Therefore, it naïvely breaks the first group across two bundles. The first bundle will contain only a `mov` operation between two `no-op` instructions. Since the RAW dependency still exists because of the problematic instructions packaged together in the second bundle, an intra-bundle stop

is forced (' `;;` '), which offers absolutely no performance benefit at all. This example stresses the importance of having a compiler smart enough. Given that the instructions are issued and executed exactly as they are packaged – since Itanium processors contain no out-of-order execution or implicit scheduling –, it is the compiler's responsibility to find the optimal combinations of bundles for maximum performance.

### 2.3.1 Itanium2 Montvale

Figure 2.4 shows the block diagram of an Itanium2 9100 series (codenamed Montvale) processor. It comprises two 64-bit cores and three cache levels per core. This architecture has 128 General Purpose Registers and 128 Floating Point (FP) registers. The specific model installed in FINISTERRAE is 9140N, which runs at 1.6 Ghz and has a 9MB L3 cache memory.

Each core has its own cache hierarchy up to the L3 level. The first level (L1) caches are 4-way set associative, 64-byte line-sized caches each and hold 16 KB of instruction or data. They are accessed in a single cycle. The data cache, which is write-through, is an integer-only memory. Therefore, FP operations bypass L1. Montvale provides a dedicated 1 MB L2 cache for instructions. This cache is 8-way set associative with a 128 byte line size and 7 cycles per instruction access. The 256 KB L2D is a write-back cache whose hit latency is 5 cycles for integer and 6 cycles for floating-point accesses. The third level (L3) is a unified, on-chip 9MB memory with the same line size than L2. Its theoretical hit latency is 14 cycles or higher.

### 2.3.2 Itanium2 Montvale PMU

Although performance counters were introduced in Chapter 1, the performance monitoring features of the Montvale Processor Monitoring Unit (PMU) deserve a separate explanation, given that its hardware counters exhibit some particular features that have been profusely used through the rest of this work.

The Montvale processor provides twelve 48-bit performance counters per thread, more than 200 monitorable events, and several advanced monitoring capabilities. The Montvale processor performance monitor architecture focuses on two usage models: workload characterisation and application profiling. To undertake any of them, performance monitors allow processor events to be monitored by programmable counters. Two sets of performance monitor registers are defined. *Performance Monitor Configuration* (PMC) registers are used to control the monitors. *Performance Monitor Data* (PMD) registers either provide data values from the monitors, or hold data values used by the PMU. As seen in Figure 2.5, Montvale provides

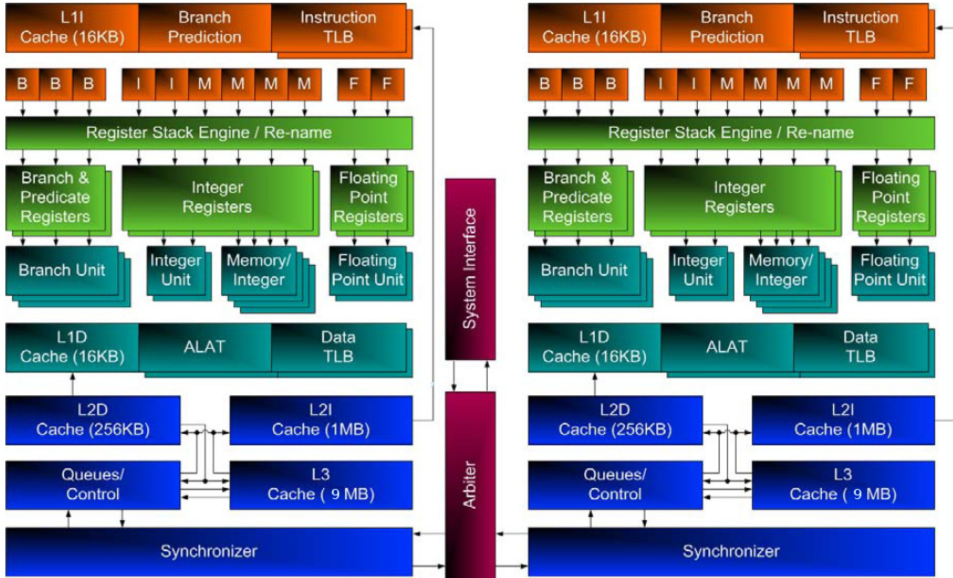



**Figure 2.4:** 1.6 Ghz Dual-Core Intel Itanium2 Montvale (9140N) architecture [2].

12 generic performance counter pairs assigned to $PMC/PMD_{4-15}$ and 4 performance counter overflow status registers ($PMC/PMD_{0-3}$). The Event Address Registers (EARs, explained in Section 2.3.3) and the Execution Trace Buffer (ETB) are controlled by three configuration registers ($PMC_{37,40,39}$). Captured event addresses and cache miss latencies are accessible to software through five event address data registers ($PMD_{34,35,32,33,36}$) and a branch trace buffer ($PMD_{48-63}$). Additionally, monitoring of some events can be constrained to a instruction address range by appropriately setting the *Instruction Breakpoint Registers* (*IBR*) and the instruction address range check register ($PMC_{38}$) and turning on the checking mechanism in the opcode match registers ($PMC_{32,33,34,35}$). Two opcode match register sets and an opcode match configuration register ($PMC_{36}$) allow monitoring of some events to be qualified with a programmable opcode. For memory operations, events can be qualified by a programmable data address range by appropriate setting of the data breakpoint registers (*DBRs*) and the data address range configuration register ($PMC_{41}$).

For a $W$ bit-width $PMD_i$ counter, an overflow interrupt occurs when the event of a counter increments thus causing carry out from bit $W-1$. The counter wraps, the overflow bit

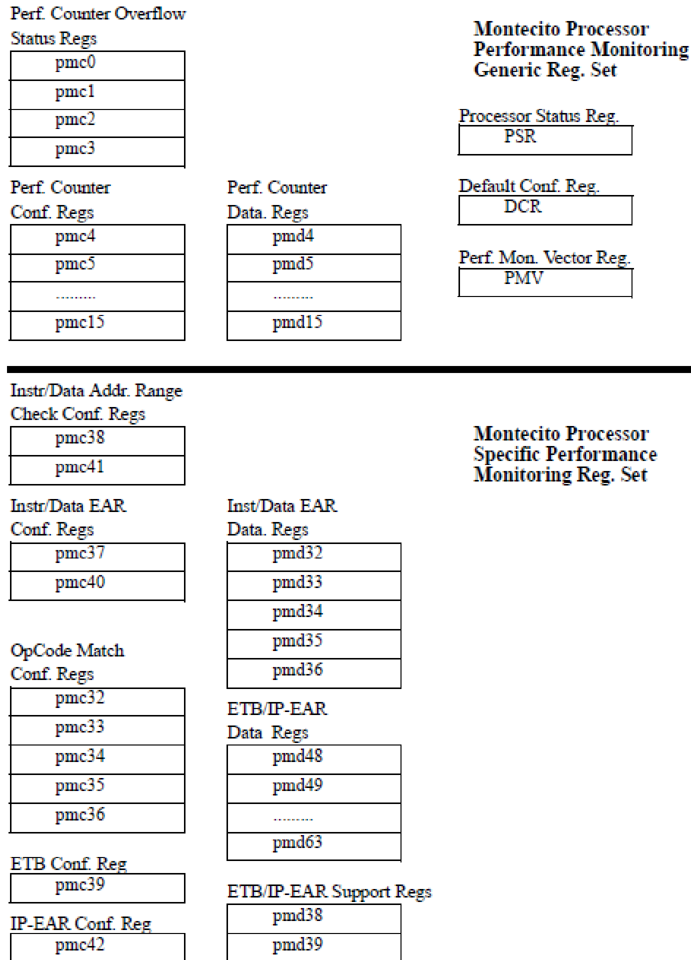



**Figure 2.5:** Montecito/Montvale Processor Performance Monitor Register Set [2].

($PMC_i.oi$) and freeze bit ($PMC_0.fr$) are set and these generate an interrupt in the PMU. Until the software does not clear those bits, no further interrupts are generated.

### 2.3.3 Event Address Registers

*Event Address Registers* (EAR) are a special type of performance counters found in all models of the Itanium2 processor. They provide event resolution for instruction and data cache misses.

Two types are available: *Instruction Event Address Registers* (IEAR) and *Data Event Address Registers* (DEAR). IEAR trigger on instruction fetches that miss the L1 instruction cache, and record the virtual instruction address and the number of cycles of the instruction that was in flight. DEAR trigger on either L1 data cache load misses, FP loads, L1 data TLB misses, or ALAT misses, and record the actual data delivery latency and the exact virtual instruction and data address at which they occur. These features allow an unambiguous localisation of given latency data and instruction accesses.

EARs enable two types of profiling: *EventBased Sampling (EBS)*, where the information is recorded at some interval expressed as a number of occurrences of an event, and *TimeBased Sampling (TBS)*, where the information is recorded at some interval expressed as a unit of time. It is possible to emulate time-based sampling using an event with a high correlation to time (e.g. number of elapsed cycles).

EBS is carried out by configuring a performance counter to count, for example, the number of data cache misses. The corresponding PMC is set up to interrupt the processor after a predetermined number of events have been observed. The DEAR repeatedly captures the instruction and data addresses of data cache load misses. When the number of misses reaches the predetermined threshold, the counter overflows, an interrupt is delivered to software, and event collection is suspended until the EAR is read by software. Afterwards, a new observation interval can be setup by rewriting the corresponding PMC.

Note that the hardware does not track all potential DEAR events, but it creates a statistical profile of cache misses of an arbitrary event resolution depending on the configured sampling period.

In this work we have focused on DEARs configured exclusively to capture cache misses, since our main goal is to perform a workload characterisation based on the latency of memory accesses, as it will be explained in Chapter 4. Figures 2.6 and 2.7 show, respectively, the PMC and PMD registers which actually comprise the Data Event Address register format. The most relevant fields are explained next. A detailed explanation of the whole EAR register set can be found in [2].

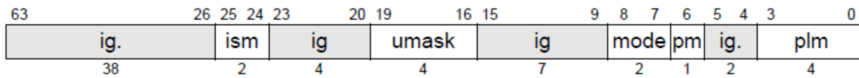



**Figure 2.6:** Data Event Address Configuration Register ($PMC_{40}$).

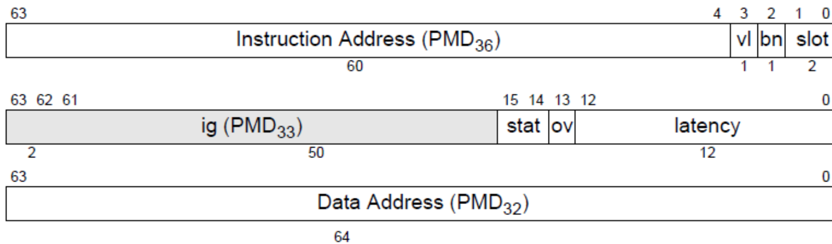



**Figure 2.7:** Data Event Address Register Format ($PMD_{32,33,36}$).

$PMC_{40}$ in Figure 2.6 can be programmed to monitor either L1 data cache load misses, FP loads, L1 data TLB misses, or ALAT misses, depending on the values set in the field `mode`. When set up to monitor L1 misses, the `umask` field is used as a power-of-two latency threshold (in cpu cycles) over which misses are captured.

Figure 2.7 shows the associated event address data registers $PMD_{32,33,36}$. When an event is captured, $PMD_{32}$ will contain the address of the data which issued the event. $PMD_{33}$ will record, among other data, the latency of that access. $PMD_{36}$ will store the address of the instruction which was in flight when the event was issued. Note that these Event Address Data registers contain valid data only when event collection is frozen ($PMC_0.fr$ is set).

To count the number of L1 data cache load misses, the pair $PMC_5/\ PMD_5$ must be used together with $PMC_{40}/PMD_{32,33,36}$. $PMC_5$ must have one of its fields configured to measure *DATA_EAR_EVENTS* events. $PMD_5$ will increment every time a L1 data cache load miss occurs until it overflows, freezing the monitoring process and raising an interruption, as explained in Section 2.3.2. $PMD_5$ can be precharged with an initial value from which to start counting so that the difference between this value and its length $W$ determines a sampling period $p$. That is:

$$pmd_precharge = 2^W - p$$

To our research, an important issue was that floating point operations bypass the L1. In fact, we confirmed that all floating points operations are considered as a L1 data cache miss. Therefore, by setting the DEAR threshold so that the counter increases in all cache misses with latency equal or higher than 5 cycles (i.e. an access to L2), we were able to detect all data accesses, not only those which caused a cache miss, thus increasing the number of captured events.

The details of programming EARs using `Perfmon` present some particularities. Given that EAR programming is not one of the objectives of this chapter, focused solely on describing the FINISTERRAE architecture, this issue will be addressed in Chapter 3 where our sampling methodology, as well as our related findings, will be presented.

## 2.4 Performance Evaluation on FINISTERRAE

The beginning of this chapter discussed the reasons why such a complex infrastructure as FINISTERRAE requires experimental assessment. Section 2.2 presented a infrastructure comprising a number of nodes in which there exist numerous factors such as several dual-core processors sharing a bus, interacting with similar buses in the same cell in a SMP configuration and, in turn, each couple of cells sharing memory in a NUMA distribution. In this context, the SMP behaviour is not obvious, the interaction of each core with its neighbour is not properly stated in the literature and it is by no means clear when bottlenecks can occur and in which cases saturation of a bus can compromise the SMP behaviour. Furthermore, dense and irregular codes behave in a different manner because of their inner nature. A dense code will access data sequentially, so it exhibits a stronger locality of references and, therefore, fewer conflict cache misses and a more efficient use of the memory hierarchy. Besides, the processor's prefetching will contribute to fetch the data before it is needed. On the contrary, a sparse code will access part, if not all, of its data non-sequentially, so the use of the cache memory will not be so beneficial. Since a good performance of many parallel scientific applications depends on the correctness of our assumptions, a study was carried out focused on quantifying the behaviour of a FINISTERRAE node depending on how the data allocation, the memory latency and the thread-to-core mapping can influence a code's final performance. Subsequent sections present this study for dense codes (Section 2.5) and sparse codes (Section 2.6).

## 2.5 Performance Evaluation of Dense Codes

### 2.5.1 Experiment setup

In addition to the load balance of the problem under study, there exist three main issues which can affect noticeably the performance of a parallel code on a rx7640 node of FINISTERRAE:

– The thread-to-core mapping: every four cores share a bus. Having several threads assigned to cores in the same bus can lead, depending on the traffic, to a performance decrease because of the competition for the bus.

– The cache coherency and consistency mechanisms require some additional traffic: applications with a high number of cache-line replacements may result in a poor performance when the data intended to keep coherent is far (another bus or even cell) or the inter-bus coherence directory must be accessed.

– The data affinity: accessing data allocated in a remote memory (that is, a memory of another cell) will take longer than accessing data in a local one. These latencies must be quantified.

The study of these issues was addressed by testing representative dense loads with appropriate thread-to-core mappings. The system was benchmarked using `Rip` [54], a suite consisting of several dense benchmarks. To manually allocate threads in cores and data in memory, the `numactl` command [55] was used.

The `Rip` suite, written in C++, was compiled with the Intel's 10.0 Linux C compiler. All the results shown in the next section were obtained using the compiler optimisation flag -O2. The system kernel is Linux 2.6.16.

### 2.5.2 Experiment #1: Influence of Thread Allocation in Bus Contention

The first issue studied was the influence of the thread allocation in relation to the cell buses. Considering that four cores share a bus (see Figure 2.2), it was reasonably foreseeable that any allocation which spreads out the threads as much as possible through different buses would get a better performance than another one which maps several threads in cores within the same bus, due to a possible bandwidth competition.

| *Cores* | *Memory allocated* | | | |
|---|---|---|---|---|
| | 10*KB* | 64*MB* | 1*GB* | 10*GB* |
| *Cell* 1 | | | | |
| 8 – 0 | 4.0 | 338.6 | 349.8 | 532.7 |
| 8 – 2 | 3.9 | 338.8 | 349.6 | 534.5 |
| 8 – 4 | 4.0 | 338.4 | 349.6 | 532.6 |
| 8 – 6 | 3.9 | 338.6 | 349.6 | 532.8 |
| *Cell* 0 | | | | |
| 8 | 3.5 | 329.0 | 340.6 | 525.7 |
| 8 – 9 | 3.5 | 352.9 | 366.4 | 546.2 |
| 8 – 10 | 3.5 | 354.3 | 366.0 | 550.2 |
| 8 – 12 | 3.5 | 342.3 | 353.7 | 539.3 |
| 8 – 14 | 3.5 | 342.2 | 353.6 | 537.4 |

**Table 2.1:** Median memory access latency (in ticks) of *memtest* benchmark for different configurations. Depending on the second core involved, Core 8 will share with it the bus, the cell or none of them. Measurements with Core 8 alone for each memory size are also shown as a reference.

To quantify this effect, a benchmark from the suite `Rip` called *memtest* was used. *Memtest* focuses on how multiple cores share the bandwidth to memory. It allocates a given-sized, private block of memory per core filled with a randomly linked pointer trail. Then, it goes through it reading and writing the data, which creates traffic associated to the data read and, subsequently, written-back to memory. To quantify the effect of sharing a bus, several configurations comprising different thread allocations were used. To avoid the system allocating the data in different memory regions (local, remote or interleaved) during the tests, all data were allocated in Cell 0. One thread was always mapped to Core 8. The other one was mapped to a core in the same cell, either to Core 9 (same socket, same bus), Core 10 (different socket, same bus), Core 12 (different socket, different bus) or Core 14 (different socket, different bus). Additionally, tests between Core 8 and some cores in Cell 1 (cores 0, 2, 4 and 6) were also performed to quantify the effect of using two cores which do not share any resources inside a cell. For comparative purposes, a test mapping just one thread to Core 8 was also carried out. Table 2.1 quantifies the outcomes of those configurations for memory blocks of 10KB, 64MB, 1GB and 10GB in clock ticks per memory access. Note that, since the RDTSC instruction was used to measure the memory access time, the final average calculation will include the access time but also some overhead. Depending on the two cores involved in each test and the size of the memory allocated, a decrease in performance was expected as long as the traffic in the bus increases (because of bigger memory sizes) when both cores share the same bus. That

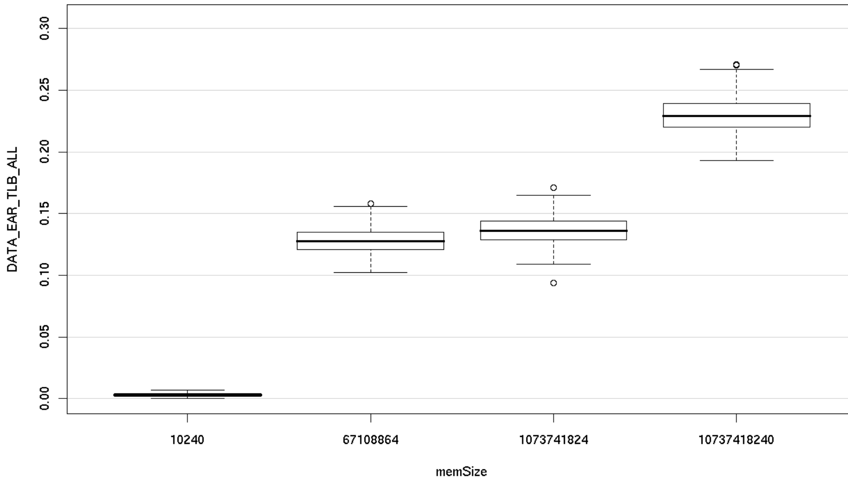



**Figure 2.8:** Average L1 DTLB misses per access for the *memspeed* benchmark. The y-axis represents the number of events. The x-axis, the memory size in bytes for the considered cases.

is, a memory size of 10KB is not expected to consume too much bandwidth since the data fits in L1 and, once loaded in cache, the bus will not be used until write-back. For block sizes bigger than 9MB (the L3 size) the traffic in the bus is expected to increase, not only because of write-back, but also because of the cache replacing. Therefore, a poorer performance is expected when two threads are mapped to two cores in the same bus and a large amount of memory is allocated.

The outcomes in Table 2.1 for 10KB show that, regardless of the pair of cores involved in Cell 0, the number of required clocks to access the data is the same (3.5 ticks). As expected, for a data block small enough to fit into L1 there is almost no traffic in the bus and, therefore, no performance differences are observed. When the second core belongs to Cell 1 the time to access the data is also almost constant (about 3.9 ticks) and higher than the previous case, as can be expected because all data are allocated in Cell 0 and must be transferred to a remote cell.

When the size of the block is increased over the L3 size (64MB and 1GB) three different cases can be identified. Focusing on Cell 0, the lowest average latency access occurs when Core 8 is alone in the bus. A second case with the highest latency access appears when Core 8 shares the bus with a core in the same processor (Core 9) or in a different socket but in the

same bus (Core 10). Indeed, a data size large enough not to fit into cache can generate enough traffic in the bus to decrease the performance when both cores compete for it. The third case appears when two cores access memory from different buses (Cores 12 and 14). In this case, the performance decrease is not as important as in the case when the bus is shared. Taking into account that the latency is not as low as the one-core case and that the throughput in the Cell Controller-to-memory bus is the same regardless of the pair of cores used, we must conclude that the Cell Controller introduces a small delay when dealing with traffic from both buses. Besides, since there are no significant differences between allocating a thread in the same socket or in other socket sharing the bus, we can also conclude that, regarding bus sharing, each core can be considered as an independent processor in this context. Focusing on Cell 1, the latency is approximately constant regardless of the core and bus involved ($\sim$338 cycles for 64MB case and $\sim$349 for 1GB) and lower than using two cores in the same cell. This upholds our conclusion about the Cell Controller introducing a certain delay.

The last case studied (10GB) shows the same three latency regions. However, the latency increases noticeably whereas the 64MB and 1GB scenarios showed little difference between them. It seemed reasonable to think that the randomly linked pointer in such a large block size was increasing the page eviction. To confirm it, we studied the behaviour of the TLB. Figure 2.8 shows the L1 DTLB misses. Indeed, we can see that the number of cache misses is similar for both the 64MB and 1GB cases. However, the 10GB case yields a higher number of TLB misses. Even if some page requests can be satisfied by the L2 DTLB, the rest will produce page faults which will increase the access latency noticeably.

We can therefore conclude that it seems advisable that the threads, regardless of the data size (assuming it is larger than the cache size), be distributed as much as possible in different buses.

### 2.5.3 Experiment #2: Influence of Thread Allocation in Cache Coherency

The second issue under study was the influence of the thread allocation upon the memory coherency protocols. The rx7640 memory coherency is implemented in two levels. A standard snooping bus coherence (MESI) protocol [56] is used for the two sockets sharing a bus, having on top of it an in-memory directory (MSI) to keep inter-bus coherence. Therefore, higher latencies are expected when the coherence has to be kept up between two cores in different buses than for two cores in the same bus since, in the former case, the directory must be read.

To quantify the effect of sharing a variable between two cores, a *producer-consumer* benchmark from the suite `Rip` was used. The producer allocates and accesses a whole data block filled with a randomly linked pointer trail, subsequently modifying the data after fetching it into cache. Once the producer has finished, the consumer just reads the whole data. We defined a configuration where Core 14 is always the producer and different cores play the role of the consumer. Table 2.2 shows the ticks per access to transfer the data from the producer to different consumers. 10KB, 128KB, 6MB and 1GB data sizes were used to make them fit in the L1, L2, L3 or in memory, respectively.

At the sight of the results we can observe that if the consumer is in the same bus as the producer the time to fetch a cache line is shorter than if both are in different buses, which is the case of Cores 12 and 15 for 10KB, 128KB and 6MB. This is due to the behaviour of the MESI protocol implemented at bus level, which is faster than accessing the directory. It is also noticeable that the time is the same regardless of whether the consumer shares the socket with the producer or not. Remembering also that cores in an Itanium2 Montvale processor do not share any cache level, we can conclude that, regarding cache coherency, each core behaves as an independent processor in this context. When the consumer is in a different bus than the producer, which is the case of Cores 0, 8 and 10, the directory must be read to check in which bus the requested data are, with the subsequent rise in the access time. Cores 8 and 10 are in the same cell, so their latencies are similar and lower than the latency from Core 0, which is in another cell. In this latter case, since the data must be brought through another Cell Controller, it exhibits the highest latency.

Despite all data fitting in any cache in the previous cases (10KB, 128KB and 6MB), it can be noticed that the time increases slightly with the data size. This fact can be explained arguing that we are observing the effect of cache collisions due to the limited associativity of the caches.

An exception to the observed outcomes occurs for 1GB. In that case, the time to fetch a cache line is practically identical regardless of the cores involved, as long as they belong to the same cell. The cause for this behaviour lies in the size of the allocated memory. For 10KB, 128KB and 6MB the data can reside in the L1, L2 or L3. However, for 1GB the producer must flush the data back to memory after modifying it, so the consumer must fetch the data from main memory in most cases instead of doing it from another cache. Note also that the time to retrieve a data from a core in a remote cell is higher than from the local memory, as it is seen in the 14-0 case for 1GB, compared to the 6MB case.

| $Prod - Cons$ | $Memory\ allocated$ | | | |
|---|---|---|---|---|
| | $10KB$ | $128KB$ | $6MB$ | $1GB$ |
| $14-0$ | 317.6 | 353.2 | 353.2 | 296.8 |
| $14-8$ | 220.0 | 258.4 | 261.2 | 194.4 |
| $14-10$ | 225.2 | 258.4 | 261.2 | 194.4 |
| $14-12$ | 79.2 | 79.2 | 87.2 | 192.0 |
| $14-15$ | 79.2 | 79.2 | 87.2 | 192.0 |

**Table 2.2:** Duration (ticks/access) of the *producer-consumer* benchmark to transfer the allocated data between two cores.

We can conclude that, to minimise the effect of cache coherency, any parallel application working with a reduced amount of shared data –not much bigger than the cache size– should map its threads to the available cores in the same bus regardless of the socket in which they are. When this is not possible, the best choice is the adjacent bus in the same cell and, as a last option, a core in a different cell. On the contrary, a parallel application which allocates a large amount of memory might saturate the bus (as shown in Section 2.5.2) with the subsequent decrease in performance. In this case, mapping the threads to cores in different buses of the same cell might be the best option since, as shown in Table 2.2, for a big amount of memory the latencies due to cache coherency become the same for all buses. Therefore, the application should firsty be characterised to find out whether the restricting factor is the traffic in the bus due to the amount of allocated memory or due to the cache coherency, in order to take a proper decision.

### 2.5.4 Experiment #3: Influence of Memory Affinity

As explained in Section 2.2, processors on a rx7640 node are arranged in a two SMP-Cell NUMA configuration. Each cell has a 64GB memory module. Therefore, data can be allocated on a local memory (threads and data are in the same cell) or on a remote memory (threads and data are in different cells). Additionally to these two modes, about one fourth of each memory module can be used to allocate data using an interleaved policy. When interleaved memory is used, 50% of the addresses are to memory on the same cell as the requesting processor, and the other 50% of the addresses are to memory on the other cell (distributed in a round-robin fashion). The main goal of using interleave memory is to decrease the average access time when accessing data simultaneously from cores belonging to different cells. Me-

mory latencies provided by the manufacturer [57] are: ∼185 ns (local memory) and ∼249 ns (interleaving memory). Latencies to remote memory are not given.

In this section, an experiment was carried out to compare the theoretical memory latency given by the manufacturer [57] with our observations. We measured the memory access latency of a small Fortran program which creates an array and allocates data in it. The measurements were carried out using EARs through the `pfmon` tool. `Pfmon`'s underlying interface, `Perfmon`, samples the application at run-time using EARs, getting the memory position and access latency of a given sample accurately.

Figure 2.9 depicts the results when allocating the data in the same cell as the used core (a), in the remote cell (b) and in the interleaving zone (c). In all cases, many accesses happen within 50 cycles, corresponding to the accessed data which fit in cache memory. There is a gap and, then, different values can be observed depending on the figure. Figure 2.9(a) shows occurrences between 289 and 383 cycles when accessing the cell local memory. The processors frequency is 1.6 Ghz, which yields a latency from 180.9 to 239.7 ns. Its average value is 210.3 ns, slightly higher than the 185 ns given by the manufacturer.

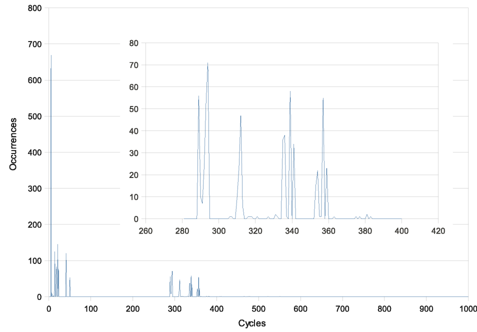
When accessing data in a remote memory we measured occurrences between 487 and 575 cycles, that is, from 304.8 to 359.8 ns, with an average value of 332.3 ns. The manufacturer does not provide any values in this case.

In the case of accessing data in the interleaving zone, the manufacturer value is 249 ns. Our measurements give two zones, depending on whether the local or remote memory are accessed. Indeed, the average access time in the interleaving zone is the average of combining accesses to the local or remote memory. Our outcomes gave an average value of 278.3 ns.

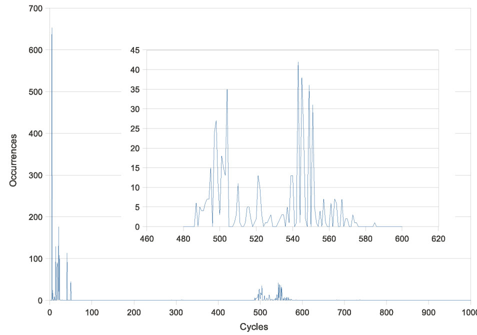
We can conclude that, when working with codes mapped to cores in a same cell (especially for those who demand a high level of cache replacement), the data should be allocated in the same cell's memory. The access to remote memory becomes very costly so, only if cores in both cells must be used, the allocation of the data in the interleaving memory makes sense.

## 2.6 Performance Evaluation of Sparse Codes

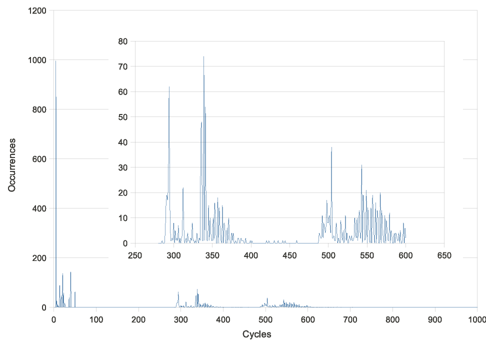
The most demanded applications in computational science are those related to simulations in physics, chemistry and biology, like fluid mechanics, weather forecast simulations, semiconductor devices, etc. In these simulations, which are often solved using iterative methods, the solution of large sparse linear equation systems is required. These methods involve working with sparse matrices, whose irregular memory access patterns can cause a scarce reuse of




(a)




(b)




(c)

**Figure 2.9:** Latency of memory accesses when the data are allocated in memory local to the core (a), in memory on the other cell (b) or in the interleaving zone (c). The y-axis shows the number of occurrences of every access. The x-axis shows the latency in cycles per memory access. Regions of interest have been zoomed in.

the data in cache memory. This fact, together with an inadequate thread and data allocation policy, often lead to a noticeable performance decrease. This section estimates the influence of these factors on FINISTERRAE for irregular codes. The sparse matrix-vector multiplication (SpMV) was chosen as a case of study in order to evaluate its performance. This kernel, central piece of numerous scientific applications and base of iterative methods for equation systems solvers, is notorious for sustaining low fractions of peak processor performance due to its indirect and irregular memory access patterns. Therefore, gaining a good understanding of the behaviour of SpMV on FINISTERRAE is very important in order to achieve a high performance.

Subsequent sections will state the prerequisites, explain the performance tests and show the results and conclusions of the experiments carried out to evaluate the data and thread allocation of irregular codes in the FINISTERRAE machine.

### 2.6.1 Experiment setup

Consider the operation $y = A \times x$, where $x$ and $y$ are dense vectors, and $A$ is a $n \times m$ sparse matrix. One of the most common data structures used for storing a sparse matrix to calculate a SpMV is the Compressed-Sparse-Row format (CSR) [58]. *da*, *index* and *ptr* are the three vectors (data, column indices and row pointer) that characterise this format. Figure 2.10 shows an implementation of SpMV using CSR storage. This implementation enumerates the stored elements of $A$ by streaming both `index` and `val` with unit-stride, and loads and stores each element of $y$ only once. However, $x$ is accessed indirectly, and unless we can inspect `index` at run-time, it is difficult or impossible to reuse the elements of $x$ explicitly. Note that the locality properties of the accesses to $x$ depend directly on the sparsity pattern of the considered matrix.

All codes were written in C and compiled with the Intel's 10.0 Linux C compiler (`icc`). OpenMP directives were used to parallelise the irregular code of Figure 2.10. All the results shown in the next section were obtained using the compiler optimisation flag `-O2`. The parallel codes use a block distribution of the sparse matrix rather than a cyclic one due to the better performance achieved (see Section 2.6.2). Tests have been performed on a rx7640 node of the FINISTERRAE supercomputer.

To perform the experiments, fifteen sparse matrices from the matrix test set presented in Table 1.1 were selected. Table 2.3 summarises their main features as well as their sparsity patterns.

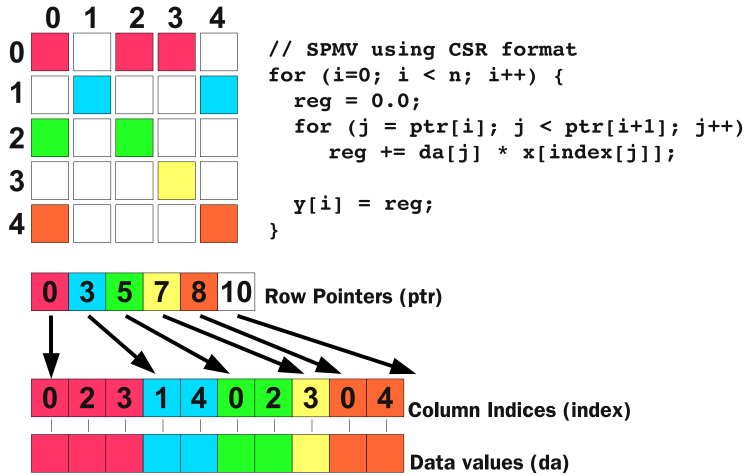



**Figure 2.10:** Compressed-Sparse-Row (CSR) format example and a basic CSR-based sparse matrix-vector (SpMV) product.

| *Matrix* | # *rows* ($n$) | # *nonzeros* ($nnz$) | $nnz/row$ |
|---|---:|---:|---:|
| av41092 | 41000 | 1683902 | 41 |
| e40r0100 | 17000 | 553562 | 32 |
| exdata_1 | 6000 | 2269501 | 378 |
| garon2 | 13535 | 390607 | 29 |
| gyro_k | 17361 | 1021159 | 59 |
| mixtank_new | 29957 | 1995041 | 67 |
| msc10848 | 10848 | 1229778 | 113 |
| nd3k | 9000 | 3279690 | 364 |
| nmos3 | 18588 | 386594 | 21 |
| pct20stif | 52329 | 2698463 | 52 |
| psmigr_1 | 3140 | 543162 | 173 |
| rajat15 | 33000 | 443573 | 13 |
| sme3Da | 12504 | 874887 | 70 |
| syn12000a | 12000 | 1436806 | 120 |
| tsyl201 | 20685 | 2454957 | 119 |

**Table 2.3:** Matrix benchmark suite used in the performance evaluation.

### 2.6.2 Experiment #1: Loop distribution

In this experiment, we evaluate the performance of the SpMV when different parallelisation strategies are applied to the code of Figure 2.10. In particular, block and cyclic distributions of loop *i* are considered.

Figure 2.11 shows the SpMV performance of both strategies using different number of threads. According to the results, we conclude that the block distribution outperforms the cyclic one. Only for matrix `exdata_1` this behaviour is not observed. We must highlight that in most of the cases the block distribution achieve higher performance than the cyclic strategy using less threads. For example, this is the case when, in Figure 2.11(a), the results corresponding to the cyclic distribution with four threads are hidden by the performance obtained with two threads and block distribution (it happens for all the matrices with the exception of `exdata_1` and `psmigr_1`). Similar behaviour is observed in Figure 2.11(b) for 8 and 16 threads.

Better results of the block distribution are due to the fact that threads work with disjoint parts of the sparse matrix when the code is parallelised using this strategy. This distribution fits better than the cyclic one in the Itanium2 architecture, where the cache hierarchy is not shared among cores (see Figure 2.4). Therefore, a block distribution is preferred over the cyclic one. For this reason, subsequent results were always obtained using a block distribution of the SpMV code.

### 2.6.3 Experiment #2: Memory affinity

The experiment of Section 2.5.4 was repeated using the SpMV code. Irregular codes are susceptible of displaying more remote accesses due to their irregular accesses (to array *x*, in this case) so it was important to find out the most adequate placement of data.

In order to guarantee the correct allocation of the data (memory affinity), the *libnuma* library [59] was used to map threads to cores (processor affinity). Next, the influence of the data allocation in the SpMV performance is studied. As an example, we show in Figure 2.12 the behaviour of SpMV code using two threads mapped to cores 8 and 12 (see Figure 2.2). Data are allocated in the memory module of the cell of cores 8 and 12 (local), in the memory of the other cell (remote), and using the interleaved policy. As is expected, an important degradation in the performance is observed when data are allocated on a remote cell with respect to local memory accesses. In particular, an average decrease of 20.7% is obtained, ranging from 3% in the best case (matrix `garon2`) to 32% (matrix `exdata_1`). Results

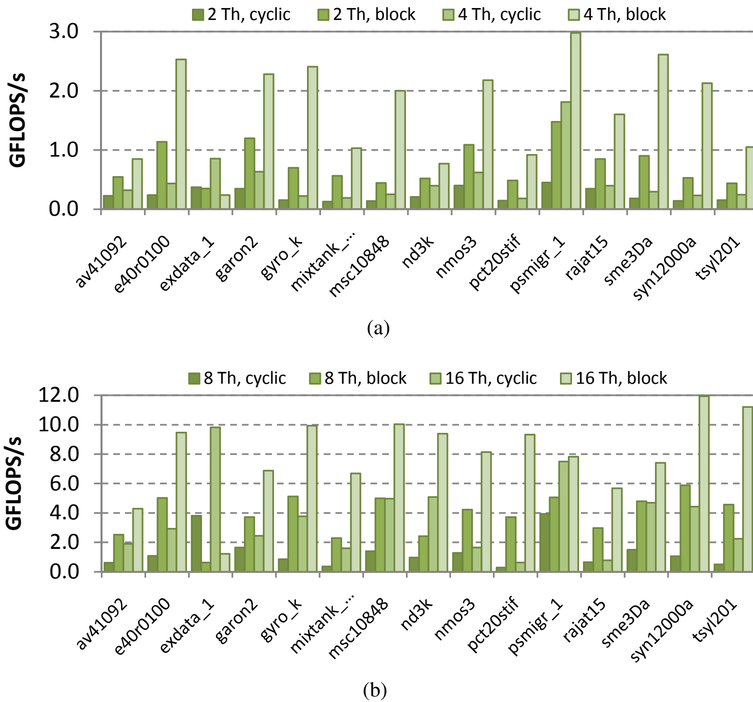



**Figure 2.11:** SpMV performance using different loop distributions: 2 and 4 threads (a), 8 and 16 threads (b).

point out that the worst behaviour is achieved for the biggest-sized matrices. When using an interleaved policy, this degradation is in average about 10%. Note that dependig on the data distribution for a particular matrix, the interleaved policy offers a performance close to local accesses (matrix `e40r0100`) or to remote accesses (matrices `nmos3` and `rajat15`). The overall behaviour analysed in this example is also observed when a different number of threads is considered.

We can conclude that data allocation has a great influence in the performance of SpMV on FINISTERRAE. When threads are mapped to cores in the same cell, it is advisable to allocate the data in the memory module of the same cell. However, if cores in both cells must be used, the allocation of the data in the interleaving memory makes sense since accessing the remote memory is very costly.

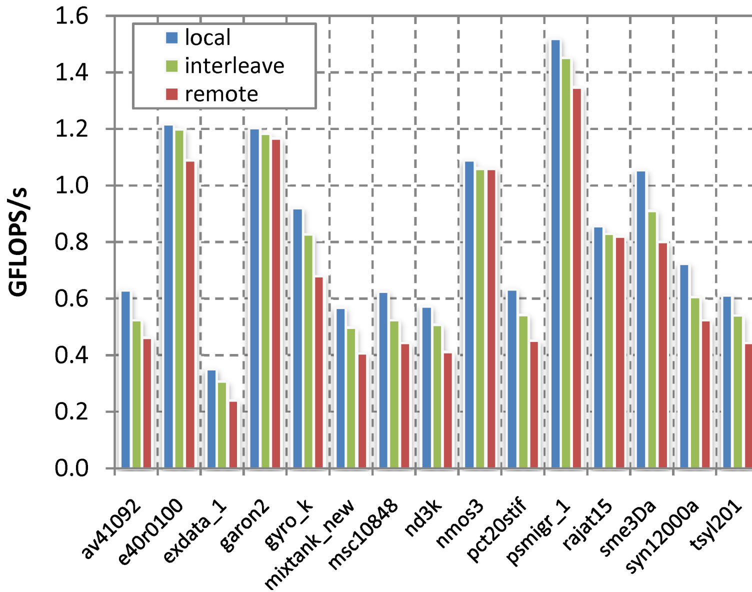
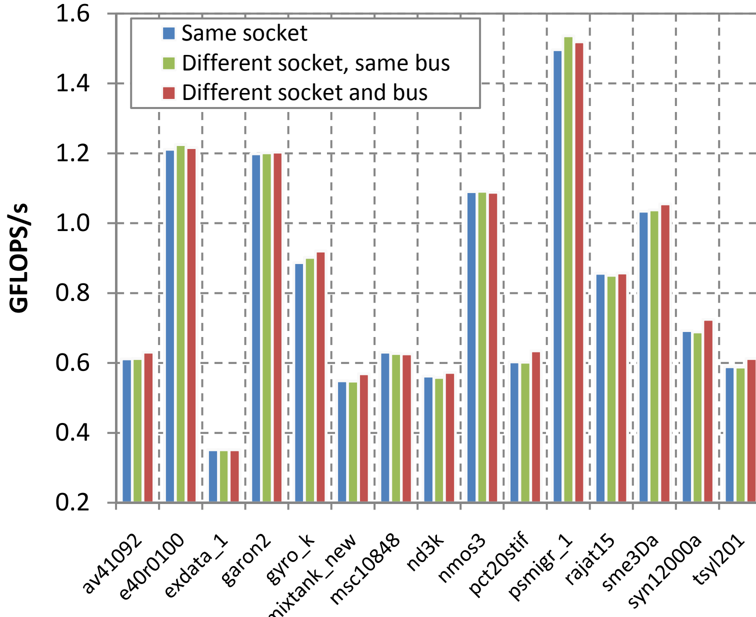



**Figure 2.12:** Influence of the data allocation on a rx7640 node.

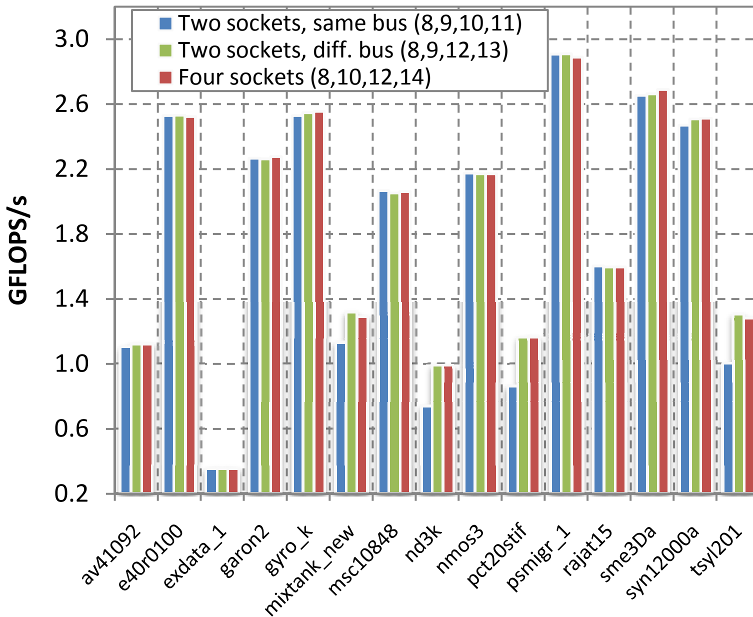
### 2.6.4 Experiment #3: Processor affinity

In this section different aspects related to the influence of thread allocation in the SpMV performance are studied. First, we have focused on evaluating the influence of mapping threads to the same processor (also called socket). Figure 2.13(a) shows the performance achieved using two threads for several mapping configurations: same socket (for example, cores 8 and 9), different socket and same bus (cores 8 and 10) and different socket and bus (cores 8 and 12). Note that data are allocated in the same cell where threads are mapped. Results point out that the influence of mapping the two threads to the same or to different sockets that share the bus is almost unnoticeable. In particular, the average performance difference is only about 0.2%. However, some improvements (up to 5%) are observed when mapping threads to cores that do not share the bus. This is particularly true in the case of big sized matrices (for example, matrices `mixtank_new` and `tsyl201`). Therefore, the contention of the memory bus has an impact on the SpMV performance.

In order to confirm the behaviour observed previously, the SpMV performance has been tested using four threads mapped to two sockets sharing the bus (cores 8,9,10,11), two sockets in different buses (cores 8,9,12,13) and four sockets (cores 8,10,12,14). Performance results




(a)




(b)

**Figure 2.13:** Influence of the thread allocation on a rx7640 node: using two (a) and four threads (b).

obtained using these configurations are shown in Figure 2.13(b). Several conclusions can be drawn.

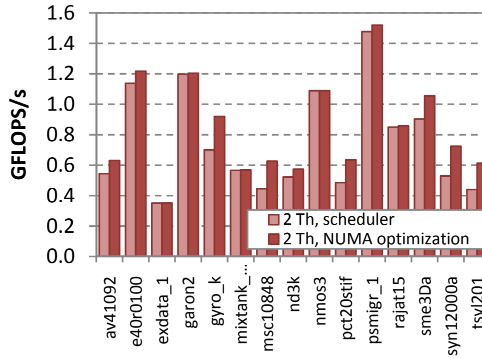
First, the same trend is observed with respect to the results using two threads (Figure 2.13(a)). That is, better performance is achieved when threads are mapped to sockets that do not share the bus (configuration 8,9,12,13). In this case, average improvement is in the range of 8% (higher than 30% for some matrices as `nd3k`, `pct20stif` and `tsyl201`). Therefore, the influence of the bus becomes more significant as the number of threads increases. Note that the average improvement is about 2% when using two threads that do not share a bus (see configuration "different socket and bus" in Figure 2.13(a))

Second, the impact of the bus for small matrices is minimal in such a way that the three considered mappings obtain similar results (for example, matrices `nmos3` and `rajat15`). And finally, there are no significant differences among using four sockets (8,10,12,14) and two sockets in different bus (8,9,12,13). This occurs because both cases there are two threads mapped to cores that share the bus.

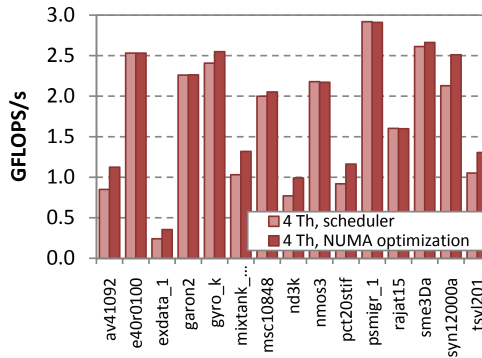
As shown above, data and thread allocation have a great impact on the performance of SpMV. Next, we will evaluate the behaviour of the operating system scheduler. With this purpose we have made a comparison among the performance achieved without NUMA considerations (naïve OS scheduler) and taking into account the particularities of the architecture (explicit data and thread allocation). The results are displayed in Figure 2.14, where only results for the best mapping configurations are shown (labeled as "NUMA optimization"). Best configurations using two and four threads were analysed before. For eight threads, we have mapped all the threads to different cores of the same cell. Results show that the OS scheduler does not take into account the NUMA particularities of the rx7640 node when mapping threads. This way, our NUMA optimisations achieve a better performance in most of the cases. For example, improvements up to 40% are observed when using two threads (matrices `msc10848` and `tsyl201`). Note that NUMA optimisation effects are more visible in the case of big matrices (for example, matrices `mixtank_new`, `nd3k` or `tsyl201`). In summary, the average performance improvement when NUMA issues are considered is 15.6%, 14.1% and 4.7% for two, four and eight threads respectively.

## 2.7 Performance model of FINISTERRAE

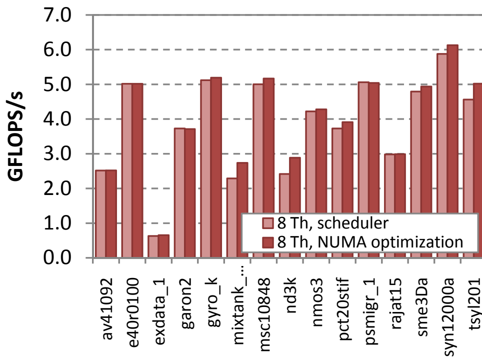
Previous sections showed that improving the performance and scalability of multicore architectures can be extremely non-intuitive, regardless of our working with either dense or




(a)




(b)




(c)

**Figure 2.14:** Effect of NUMA optimisations on a rx7640 node.

sparse codes. Although there exist some stochastic analytical models and statistical performance models which can accurately predict performance, they rarely provide insight into how to improve the performance of programs, compilers and computers. Besides, they are usually difficult to use by nonexperts. Instead of using any of them, we approached this problem by developing a Roofline Model for FINISTERRAE.

The Roofline Model [60] provides realistic expectations of performance and productivity. Introduced by Williams and Patterson at Lawrence Berkeley National Laboratory, it does not try to predict program performance accurately. Instead, it integrates in-core performance, memory bandwidth, and locality into a single readily understandable performance figure, showing inherent hardware limitations for a given computational kernel and showing potential benefit and priority of optimisations. In addition, the model can also be used for guide tuning, as long as information in run-time about the current performance of a given kernel can be obtained. In this regard, hardware counters can assist in this task. Indeed, Williams holds that *"there seems to be a synergistic relationship between performance counters and the Roofline Model."*.

Following sections will introduce the Roofline Model, the process of development for FINISTERRAE and the main results achieved.

### 2.7.1 Roofline Model Construction

The Roofline Model relies on three metrics: *Computation* measured as floating point performance (GFlops/s), *Communication* measured as DRAM bandwidth (GB/s) and maximisation of *Locality* to minimise communication. The metric that relates performance to bandwidth is defined as *Operational Intensity*. It is measured in $Flops/Byte$ and means "FP operations performed per byte of DRAM traffic transferred". That is, traffic is measured between the caches and memory, not between the processor and the caches. This measure predicts the DRAM bandwidth needed by a kernel on a particular computer.

Figure 2.15 shows the roofs of our model for FINISTERRAE. The plot is on log-log scale. The Y-axis shows the attainable double performance in GFlops/s [2]. The X-axis displays the Operational Intensity. The horizontal line shows the peak floating-point performance of the computer, and its computation values can be derived either from the processor's manual or from performance benchmarks. The maximum attainable bandwidth is a line of unit slope

[2] Diverse sources use either the term *Flops* or *Flops/s* to denote the same magnitude, namely "number of floating point operations per second". In this dissertation we will use the nomenclature *Flops/s*

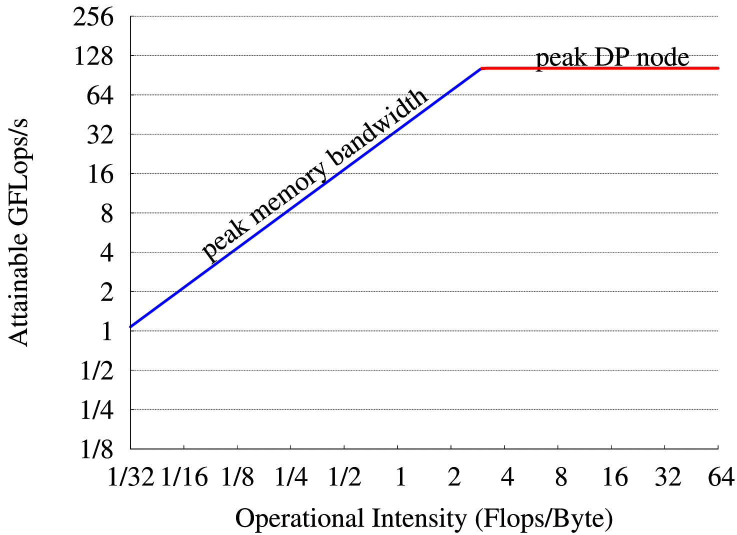



**Figure 2.15:** Roofs of FINISTERRAE Roofline Model.

($\frac{GFlops/s}{Flops/Byte} = GBytes/s$ ) and can be derived also from the architecture's manual or from a benchmark. Both lines intersect at the point of peak computational performance and peak memory bandwidth.

Both the peak floating-point performance and maximum bandwidth of FINISTERRAE were obtained from the manufacturer's documentation [57]. Note that the former (102.4 GFlops/s) comes from the product of a Montvale cores's peak performance times the number of cores in a FINISTERRAE node, whereas the latter (34.4 GBytes/s) comes from the maximum bandwidth of a memory bus times the number of buses in a node (see Figure 2.1). The attainable performance of a given kernel is upper bounded by both the peak flop rate, and the product of bandwidth and the flop:byte ratio.

$$GFlops/s = min \begin{cases} Peak\ GFlops/s \\ Peak\ Memory\ BW * actual\ flop : byte\ ratio \end{cases}$$

These roofs cannot be ever reached, since they are physical limits given by the architecture. Note that these roofs are created once per multicore computer and can be reused for any kernel.

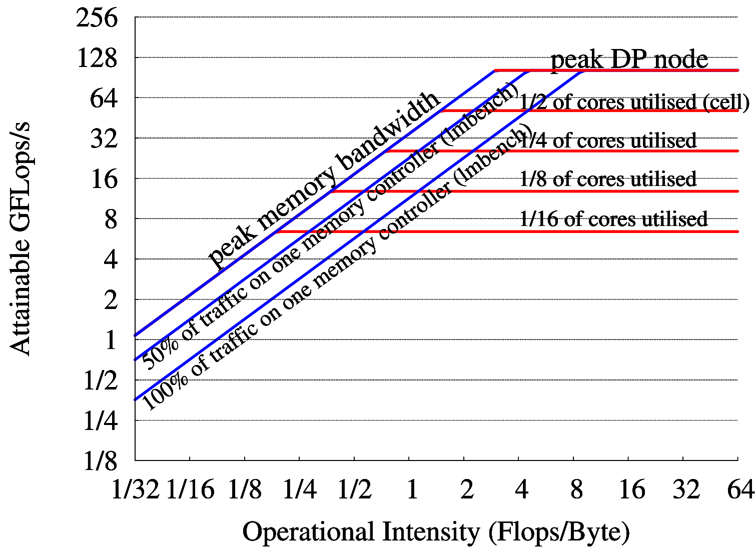



**Figure 2.16:** Ceilings added to the FINISTERRAE Roofline Model.

The next step consists in adding *ceilings* to the model. Ceilings are performance barriers which limit the maximum attainable performance. They suggest which optimisations to perform and the potential benefit to achieve. Note that we cannot break through one ceiling without first performing the corresponding optimisation. Figure 2.16 depicts computational (horizontal) and bandwidth (slanted) ceilings. These ceilings are not fixed, and can be chosen depending on the performance limits we are interested in. Should a single processor be under study, some metrics of interest could be the Mul/Add imbalance, lack of Instruction-Level parallelism, or any other which can limit the performance of a processor. Our study is focused on a FINISTERRAE node comprising several processors. Therefore, computational ceilings in the figure show performance limits depending on the number of cores involved. These values are derived from the architecture's manual [57]. Bandwidth ceilings are related to memory imbalance and were collected by a tuned version of the LMbench benchmark [61]. Each ceiling denotes the maximum sustained bandwidth attainable when all the traffic is concentrated in a single cell (ie, must be handled by a single memory controller) or when it is distributed between both cells. Ceilings, as well as roofs, are measured only once per multicore computer and can be reused for any kernel which runs in that machine.

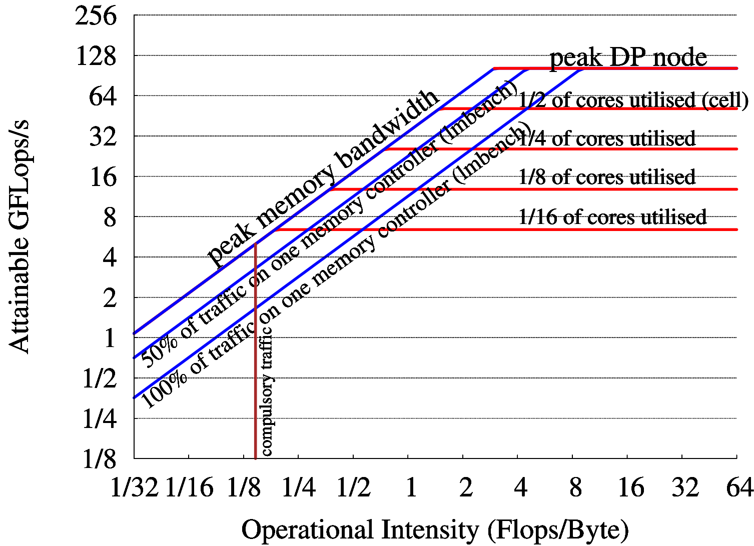



**Figure 2.17:** Locality walls for the SpMV-FINISTERRAE combination.

The last step to create the Roofline Model involves the computational kernel under study. The attainable performance for a kernel is related to its operational intensity. Indeed, moving to the right of the Roofline Model (towards higher values in the x-axis) means a higher number of FP operations per byte transferred from memory to cache and, therefore, a better performance. There is a limit in the maximum operational intensity of a kernel, given by the lower limit to communication: the compulsory traffic. This limit is called *locality wall* and is unique for a kernel-architecture combination. It must also be taken into account that the actual operational intensity may be lower due to cache misses. A kernel can be cpu-bound or memory-bound depending on whether its maximum operational intensity is on the right of the ridge point or on the left, respectively.

Figure 2.17 shows the locality wall for the SpMV. This limit was obtained by reducing the size of the matrix $A$ and the arrays $x$ and $y$ until they fit into cache, narrowing down the accesses to memory to the compulsory traffic to fetch the data once into cache. As seen in the figure, the SpMV is a memory-bound kernel, with a maximum operational intensity rather low due to its scarce reuse of the data in the cache memory.

Next sections present the results of the parallel SpMV on the FINISTERRAE Roofline Model.

### 2.7.2 Experiment setup

Once sketched the roofs and ceilings of the Roofline Model for FINISTERRAE, the next step consisted in measuring the performance of a program to depict it according to the magnitudes of this model. Our objective was to see graphically the influence of thread and data allocation studied in Sections 2.5 and 2.6 in a way that could cast some light about which improvements might be made.

To keep on going with our study of irregular codes, we used the same parallel SpMV using block distribution of previous sections. To place a performance point in the Roofline Model, a pair of coordinates (*[Operational Intensity, Attainable Performance]*) are needed. In order to get them, PAPI [62] was used to instrument the code and access native events of the Montvale processor, measuring the following magnitudes:

**Computation *(GFlops/s)*:** The number of FLOPS of the kernel is given as the sum of the values per thread returned by the event `FP_OPS_RETIRED`. The elapsed time is given by the function `PAPI_get_real_usec()`. The Computation will be calculated as the sum of all FLOPS divided by the time of the slowest thread.

**Operational Intensity *(Flops/byte)*:** The traffic transferred between main memory and cache memory is measured in Montvale as the sum of all bus memory transactions, stated as the number of full cache line transactions (event `BUS_MEMORY_EQ_128BYTE_SELF`) added to the number of less than full cache line transactions (event `BUS_MEMORY_LT_128BYTE_SELF`) [2]. The number of bytes transferred per thread is then calculated according to the following formula:

$$\begin{aligned} Bytes\ transferred = & BUS_MEMORY_EQ_128BYTE_SELF \times 128 \\ & + BUS_MEMORY_LT_128BYTE_SELF \times 64 \end{aligned}$$

The whole number of bytes transferred by the kernel will be the sum of the number of bytes per thread. The value of the Operational Intensity is calculated as the quotient between the sum of FLOPS from all threads and the sum of bytes transferred by all threads.

| *Matrix* | 2*th* | 4*th* | 8*th* | 16*th* |
|---|---|---|---|---|
| *pct*20*stif* | 0.939 | 0.882 | 0.801 | 0.703 |
| *exdata*_1 | 0.005 | 0.002 | 0.002 | 0.002 |

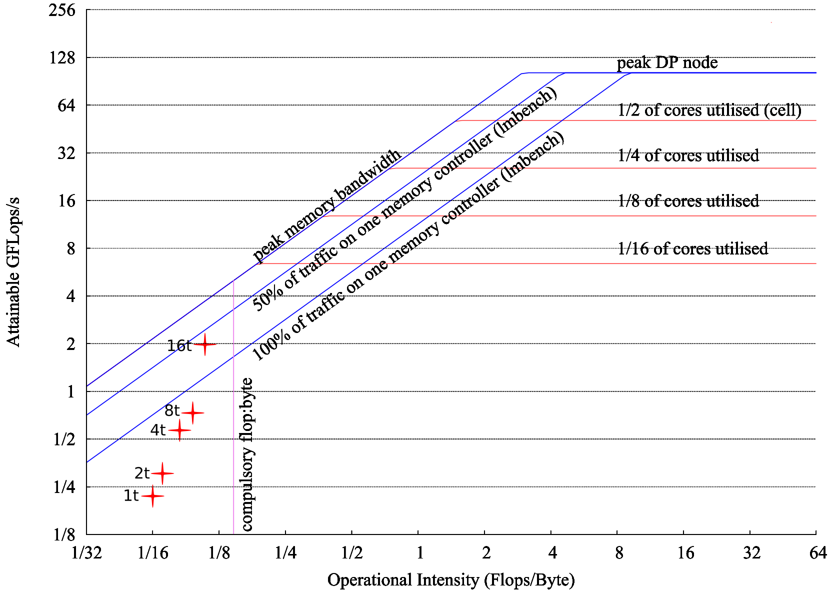
**Table 2.4:** Imbalance of matrices `pct20stif` and `exdata_1`.

The matrix test set from Table 2.3 was represented using the roofline model. The following sections will analyse the results for two of those matrices, `pct20stif` and `exdata_1`, chosen as being paradigmatic of a good and a badly-balanced matrix, respectively. Quantitative data about imbalance are shown in Table 2.4. It shows the imbalance as a result of dividing each matrix in blocks for 2, 4, 8 and 16 threads. This value is given as the quotient of the block with the lowest NNZ value and the one with the highest NNZ value. The closer the value to 1, the better the balance. It seems clear, therefore, how `pct20stif` is much better balanced than `exdata_1`.

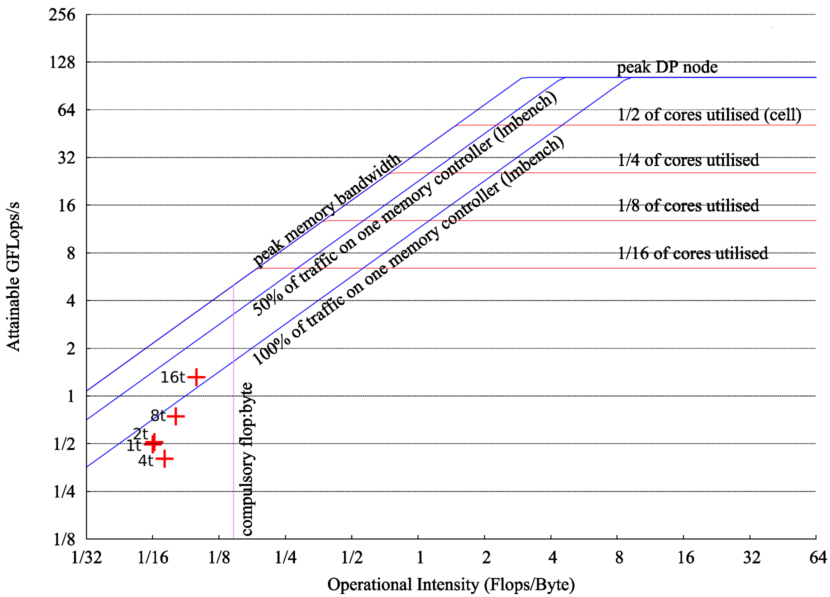
### 2.7.3 Experiment #1: Naïve parallelisation

In this configuration, the parallel SpMV was executed for 1, 2, 4, 8 and 16 threads. The Linux scheduler was allowed to map threads to cores at its will. Data were allocated by the default system first-touch policy. Figure 2.18 shows the performance of the parallel SpMV for matrices `pct20stif` (a) and `exdata_1` (b). Each circle refers to the whole performance of each n-thread case. Note that `pct20stif` shows a much more regular pattern than `exdata`, which concentrates most of its nonzero values into a small region. Some interesting information can be inferred from both Roofline Models:

– Figure 2.18(a) shows performance points that grow both in computation and operational intensity as the number of processors increases. Indeed, given the big size of the matrix, the 1-thread case will show a high number of conflict misses, which decreases the ratio flops:byte (i.e. the operational intensity) with respect to the maximum value (the compulsory misses wall). Consequently, the number of GFlops/s is not as high as it could be if the conflict misses were lower. As the number of threads increases, the matrix (and the arrays $x$ and $y$) is shared out among the processors. Therefore, the number of conflict misses decreases, the points in the graph shift towards the compulsory wall, and the computation value increases.




(a)




(b)

**Figure 2.18:** Roofline Model of SpMV for matrices `pct20stif` (a) and `exdata` (b).

– Note that, whereas the number of processors duplicates in each case, the points in the graph are not equidistant. There is a bigger gap between 2 and 4 processors, and between 8 and 16, than the remaining cases. Quantifying it, the ratio of attainable GFlops/s between 2 and 1-thread cases is $\sim 1.35$. So it is with the ratio between the 8 and the 4-thread cases. However, the ratio between the 4 and the 2-thread cases is $\sim 1.9$, and between 16 and 8 threads is $\sim 2,7$. Two causes can be found here. Firstly, the number of conflict misses does not decrease linearly as the number of processors increases. Thus, the shift increment of each point towards the compulsory wall is not constant. Secondly and more important, we do know that the SpMV uses its master thread to allocate all data before starting the computation. Therefore, the system's first touch policy will place all data uniquely in the master's memory. When using 2 threads, they are bound to have been attached to cores in different cells. One of them will need to fetch data remotely and this explains the little difference in performance between 1 and 2 threads. However, for the 4-thread case, the scheduler seems to have mapped, most of the threads to cores in the same cell and, therefore, the performance is much higher. Again, for a 8-thread case, those are spread out between both cells, and the difference in performance to the 4-thread case is not noticeable. For 16 threads all cores are used, although 8 of them will fetch data from a remote cell. Therefore, performance will be higher than for 8 threads, but not as high as the architecture allows (the 16-thread point is far from the peak memory bandwidth).

– Whilst `pct20stif` is a matrix with a common pattern with diagonal shape, `exdata_1` presents a very particular one, where most of its data are concentrated in a small region. Hence, this pattern is prone to cause load balance problems. The representation of its performance in the Roofline Model, as shown in Figure 2.18(b), substantially differs from the one of `pct20stif`. The performance achieved is virtually identical for 1-thread and 2-thread cases. Indeed, OpenMP divides evenly the number of rows among the available threads. Splitting this matrix in two halves gives $\sim 99.6\%$ of the load to one thread (in the 2-thread case). Therefore, a glance to the figure attracts our attention to an important load imbalance problem.

– The 4-thread point for `exdata_1` is placed below the points for 1 and 2 threads and slightly to the right of them. This means that, whereas this case yields a worse performance, its operational intensity is better. Therefore, the load is badly balanced but the bus bandwidth is less saturated than previous cases. So we can conclude that the

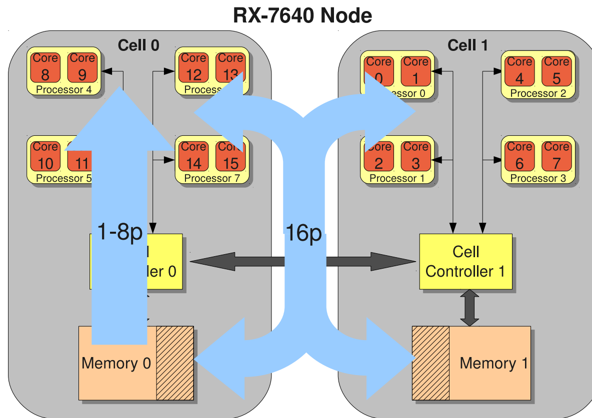



**Figure 2.19:** Experiment #2 setup.

scheduler spread out two threads to each cell, but keeping most of the load in the same cell.

– 8 and 16 threads provide a finer distribution of the matrix among the threads in both cells, so the performance increases in both GFlops/s and operational intensity.

### 2.7.4 Experiment #2: Exploiting thread allocation

This experiment executes the SpMV for 1, 2, 4, 8 and 16 threads. Threads were mapped to cores explicitly using the `sched_setaffinity()` method and data were allocated manually to memory modules using the Linux `numactl` command. Whereas in the previous section all threads were mapped to cores by the Linux scheduler and data were allocated in Cell 0 due to the system's first-touch policy, in this experiment we made sure that threads 1 to 8 were mapped to cores in Cell 0, as well as their data (see Figure 2.19). Specifically, the allocation sequence was *"8-12-10-14-9-13-11-15-0-4-2-6-1-5-3-7"*, which means that, for 1 thread, it was mapped to Core 8. For a 2-thread case, one thread was mapped to Core 8 and the other one to Core 12, and so on. Note that all threads are placed as far as possible from the remaining ones in order to spread the threads out among the available buses. For 16 threads, data were allocated in the interleaving zone, and threads mapped according to the sequence given above. These distributions try to take advantage of the knowledge acquired and presented in Sections 2.5 and 2.6.

Figure 2.20 shows the outcomes for matrices `pct20stif` and `exdata_1`. Note the distribution of points for cases 1p to 8p in Figure 2.20(a). Unlike the previous section, in this case all the performance values are vertically equidistant from each other. We know from Table 2.4 that `pct20stif` is a well-balanced matrix. Therefore, the only remaining cause for imbalance would be the thread allocation. However, in this case threads have been mapped manually to cores in the same cell where data are, taking care of keeping them always balanced between both buses in the cell. That justifies the well-distributed points in the Roofline Model.

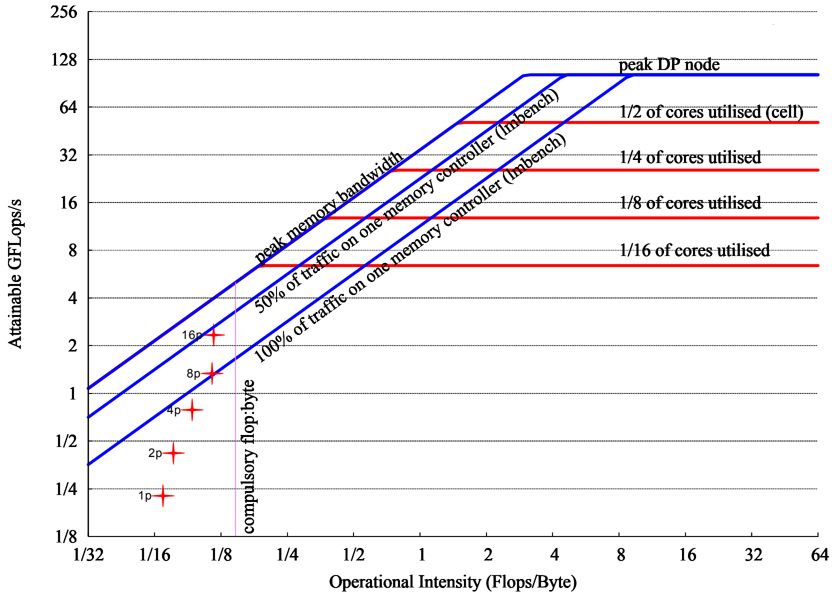
16 threads is a particular case. We noticed an increase in performance as steady as in previous cases. However, the Operational Intensity is almost alike. In this case, a decrease in performance was expected because data were allocated in the interleaving memory, which presents a latency higher than local data. A rough calculation will show the explanation for this fact: data fit into cache for both 8 and 16-thread cases. Taking into account that matrix $A$ comprises 3 arrays of the size given between brackets: $da(NZ)$, $index(NZ)$ and $ptr(N+1)$, then the memory needed to allocate $A$ together with arrays $x(N)$ and $y(N)$ is:

$$\begin{aligned} 2269501\ NZ \times 16\ bytes/double \times 2 = &\ 69.26\ MBytes \\ (52000+1) \times 16\ bytes/double = &\ 0.79\ MBytes \\ 52000\ N \times 16\ bytes/double \times 2 = &\ 1.59\ MBytes \\ Total = &\ 71.64\ MBytes \end{aligned}$$

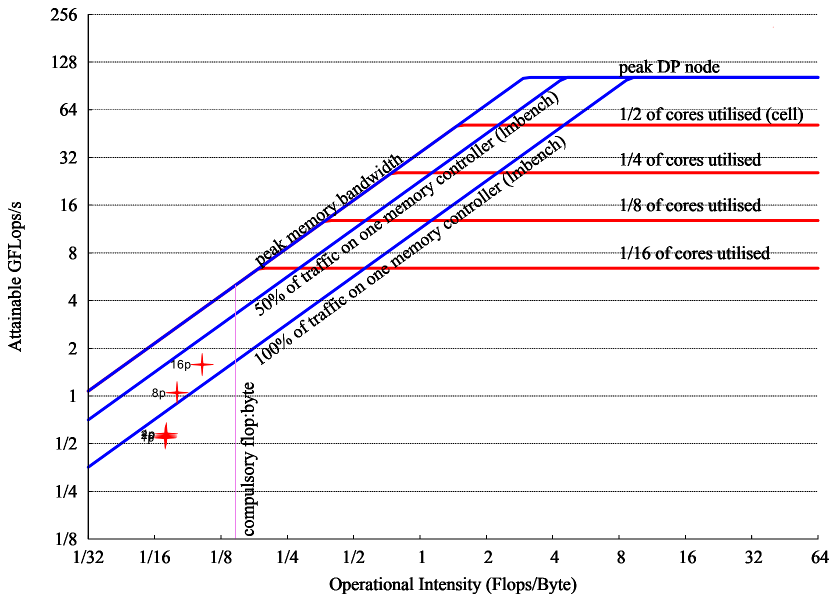
This size, divided by either 8 or 16 processors, yields a value below the 9 MBytes size of the L3. Therefore, although the higher latency will influence the performance, only the compulsory cache misses together with a limited amount of conflict misses (which prevents the performance points to reach the compulsory wall) will occur. That is the reason why the operational intensity is similar in both cases.

Figure 2.20(b) shows the Roofline Model for matrix `exdata_1`. As expected, the only difference in performance with Figure 2.18(b) is the 4-thread case. The manual thread allocation in Cell 0 balances better the data among the buses in the cell. However, the imbalance due to the irregular nature of the matrix still exists, and that is why the performance is practically the same for 1, 2 or 4 threads.

This experiment upholds our conclusions from previous sections which stated that codes with a high level of cache replacement and data size larger than the L3 cache should better have their threads shared out among the available buses and their data placed in the same cell.




(a)




(b)

**Figure 2.20:** Exploiting thread distribution of SpMV (experiment #2) for matrices `pct20stif` (a) and `exdata` (b).

Besides, this confirms the usefulness of the Roofline Model to give information at a glance about the performance of an irregular code related to load balance issues.

## 2.8 Conclusions

This chapter has presented the architecture of the FINISTERRAE supercomputer and evaluated its performance with both dense and sparse codes. Results show that, especially for applications that use the bus intensively, the effect of sharing a bus between two or more cores degrades noticeably the performance and should be avoided by spreading the threads out among cores in different buses when possible. A noteworthy fact is that every core in the same processor behaves as an independent processor, so we can consider two processors in a bus as four independent cores.

Regarding the cache coherency, the effects in the performance are noticeable when dealing with small sizes of data which fit into cache memory. The more the memory requirements increase, the less significant the effect is. Therefore, for small data sizes, it will be advisable to map the threads to cores in the same bus, so that data can be kept coherent without accessing the directory. Conversely, for larger data sizes, the negative effect of sharing a bus will be more important and it will mask the effect of cache coherency. Indeed, this will be the typical scenario found, where a computational problem using sparse codes handles data size much larger than the cache size.

It seems clear that the ideal situation would be the one where data of each thread are independent from each other, so threads can be allocated far from each other without taking care of coherency issues. It is in this case where sparse reordering techniques, working together with sampled data, may be beneficial. Our research involving these techniques is presented in Chapter 4.

Another problem relates to the memory affinity and the effect of the NUMA factor (i.e. remote to local memory latency ratio). It has been shown that the local and remote memory access latencies yield important differences between them. Therefore, in cases where the threads must be spread in two cells the best policy would be to analyse and split the data in both memory zones, maximising the locality or, when this is not possible, allocating the data in the interleaving zone. An automatic way of attempting to carry this process out is performed by the operating system. It provides a first-touch policy which allocates data pages in the local memory of the thread which first accesses any data. This is useful in those cases where each thread in a parallel program accesses its own data exclusively. However, many OpenMP

programs have all their data previously accessed by a master thread, before spawning several threads. In that case, all data will be allocated in the local memory of the master thread, regardless of the cell where other threads are allocated, with the subsequent performance decrease. This is one of the scenarios where a page migration policy will prove to be specially useful, as it will be discussed in Chapter 5.

Finally, this chapter showed the development of a *Roofline Model* for FINISTERRAE, which provided us with an insightful way to confirm at a glance our previous observations regarding thread and data allocation, as well as to suggest in which way improvements must be addressed.

# 3

# Accessing Hardware Counters on Itanium2 Montvale. The Perfmon interface

## 3.1 Introduction

Up to this point in this dissertation, hardware counters have only been indirectly accessed using high-level libraries such as `PAPI` [62] or command-line tools like `pfmon` [63]. A deep study of the possibilities offered by the hardware counters on FINISTERRAE, our Intel Itanium test architecture, requires the adoption of a lower-level interface which enables us to access all their features (goal G2-HWCSTUDY). To do so, a study of the available Linux monitoring interfaces has been carried out.

There are two levels of monitoring commonly used nowadays:

– *Program-level monitoring* collects information such as basic block call counts. The information is collected by instrumenting the program. This is achieved at compile time or dynamically by tools like PIN [64] or Paradyn [65], for instance.

– *Hardware level monitoring* collects information at the micro-architectural level such as the number of caches misses. It requires hardware support in the processor. This is typically implemented by the Performance Monitoring Unit (PMU) which exports a set of programmable counters. Monitored programs do not need to be recompiled or modified to collect the performance information.

This dissertation relies on hardware-level monitoring. As well as the functionalities of each PMU model can vary greatly, monitoring tools have very different needs depending on what they measure. For instance, some tools collect simple counts while others collect profiles. Some tools operate on a per-thread basis while others measure on a system-wide basis,

i.e., across all threads and possibly across multiple processors. Therefore, there are a wide variety of monitoring tools. Some details of the most relevant Linux monitoring interfaces are listed next:

- **OProfile [66]:** It is a system-wide profiler, capable of profiling all running code at low overhead. It consists of a kernel driver and a daemon for collecting sample data, and several post-profiling tools for turning data into information. It is supported on all major platforms.

- **Perfmon [67]:** It is a generic, flexible kernel interface to access performance counters of all major processors (Itanium, X86, MIPS, Cray, Sun SPARC, etc.). This interface makes it possible to count and sample on a per-thread and system-wide basis. Unlike other interfaces, it can access all the model-specific advanced features of recent PMU models such as Itanium2 Branch Trace Buffer (BTB) and Event Address Registers (EARs).

- **Perfctr [68]:** It is provided as a separate kernel patch. It supports per-thread and system-wide monitoring for most major processor architectures, except for Itanium.

- **VTUNE [69]:** Performance analyzer that uses its own kernel interface which is implemented by an open-source device driver. The interface supports system-wide monitoring only and is very specific to the needs of the tool.

- **HPCPI [70]:** It is a low-overhead statistic sampling profiler. It can display data at multiple levels of granularity: binary file, procedure and code line.

- **HP Caliper [71]:** This is a professional tool which works with all major Linux distributions for Itanium processors. It collects counts or profiles on a per-thread or per-CPU basis. It exploits all the advanced Itanium PMU features, such as the BTB and EARs. The profiles are correlated to source and assembly code.

- **PAPI [62]:** Already used in Chapter 2, it is a high level API that provides a platform (OS and processor) independent programming interface. It achieves OS-independence by providing a layer of abstraction over the interface provided by kernel extensions that provide access to counters. Its processor independence is given by a set of high level events available on specific processors.

- **Perf_counters [72]:** Performance Counters for Linux (PCL) is a kernel-based subsystem that provides a framework for collecting and analysing performance data. It has been included in the mainline Linux kernel since version 2.6.31, precluding other more mature solutions from being included instead [73]. It lacks support for the Itanium architecture.

From those options, the one which has been used through the rest of this dissertation is `Perfmon`, based on the following criteria:

- `Perfmon` was initially implemented for the Itanium architecture by one of the Itanium Linux kernel designers and subsequently extended to support all the x86 family processors. Therefore this tool can be considered to be reliable for this architecture.
- Despite other interfaces which monitor on a system-wide basis uniquely, `Perfmon` makes it possible to count and sample also on a per-thread basis. This is a key feature, since this dissertation considers parallel irregular codes with a thread-level granularity.
- `Perfmon` provides a low-level API which gives control about the whole monitoring process, particularly the Event Address Registers (EARs).
- Its kernel-level support minimises the sampling overhead [74].
- It is distributed under an open-source license, which enables us to inspect and modify the code when required.
- It provides a command-line tool, `pfmon`, useful to perform preliminary monitoring tests.
- There is an active community of users and developers who contribute code to solve bugs and support numerous processors.
- Other high-level tools (such as `PAPI`, used also in this dissertation) use `Perfmon`.

Additionally, a library called `libpfm` is supplied together with `Perfmon`. `libpfm` is a helper library that permits applications to program a PMU. Since PMUs from different processors differ, `libpfm` provides an abstraction layer to homogenise its use. The library provides a simple translation service whereby a user specifies an event to measure and the library helps to figure out the parameters and PMU registers needed to program the PMU.

To do the actual PMU programming (that is, to write and read the PMC and PMD registers, introduced in Section 2.3.2) `Perfmon` must be invoked.

The version of the library and the interface used in this dissertation are `libpfm 3.9` and `Perfmon 3`, respectively. A patched 2.6.29.6 vanilla Linux kernel was used.

A `Perfmon` session describes the typical sequence of actions necessary to collect measurements, which can be summarised as follows:

1. Create a `Perfmon` context
2. Program the PMU
3. Start monitoring
4. Run the code to measure
5. Stop monitoring
6. Read results
7. Destroy the `Perfmon` context

The interface can be used in two modes: to *count* the exact number of events occured (e.g. number of cache misses) or to *sample* events (e.g. memory addresses of events which caused L1 misses sampled at a given rate). In addition, codes can be *self-monitoring* (the monitored code itself includes the monitoring sequence of actions) or *non self-monitoring* (the program which monitors is different from the monitored one). These modes are explained in more detail in Section 3.2.

This chapter is organised as follows. Section 3.2 introduces, from a programming perspective, how to access hardware counters using `Perfmon`. To evaluate its performance and reliability in our test platform, Section 3.3 performs a set of tests to evaluate `Perfmon` with sparse and dense codes on FINISTERRAE. The conclusions of this study are drawn in Section 3.4.

## 3.2 Perfmon programming

There exists a staggering number of available events to capture depending on the processor model. While most of them can be both counted and sampled, there are a few which were

conceived to be exclusively sampled. This is the case of EAR events. EARs, introduced in Chapter 2, are designed to create a statistical profile of a given event. The majority of the work performed in this dissertation has been carried out using statistical profiling. Hence, this section focuses on describing the EAR sampling process using `Perfmon`.

The process followed to profile a program is described in the statechart shown in Figure 3.1: before the program to be monitored runs on a processor, a profiler has configured the processor PMU to sample occurrences of a given event. Then, the program starts running and, eventually, it will generate enough instances of an event to make the related PMU hardware counters overflow. At that moment, an interruption is raised, an interrupt handler collects the data and reprograms the PMU to keep on sampling the same event or a different one, and the collected information is sent to the profiler. In case that the profiler and the monitored program are the same one, it is a self-monitoring program. Note that the sampling process is the same regardless of using EARs or any other type of event; only the kind of information collected changes.

Sampling with `Perfmon` can be undertaken either using a sampling buffer or not. When no sampling buffer is used, every time a counter overflows the data collected at kernel level must be made available at user level so that the Profiler can read them. This process involves some overhead related to the interruption handle. The addition of a buffer enables `Perfmon` to accumulate the outcome of several overflows before copying it at user level, reducing therefore the overhead. Hence, the use of a sampling buffer is recommended.

During the sampling process, the PMU state includes the values of the PMC and PMD registers, as well as other related registers. The PMU state, along with the associated software state, is called a `Perfmon` *context*. This is the mechanism whereby the programmer can access the collected information.

The basic procedure to configure a `Perfmon`-based code to sample a multithreaded application using a buffer is as follows:

1. A `Perfmon` context is allocated.
2. Some basic parameters are configured: sampling period, event to monitor, PMCs and PMDs to use, monitor in kernel and/or user level.
3. A sampling buffer is allocated and attached to the context.
4. An interrupt handler is set up to manage the interruptions raised when the buffer is full.

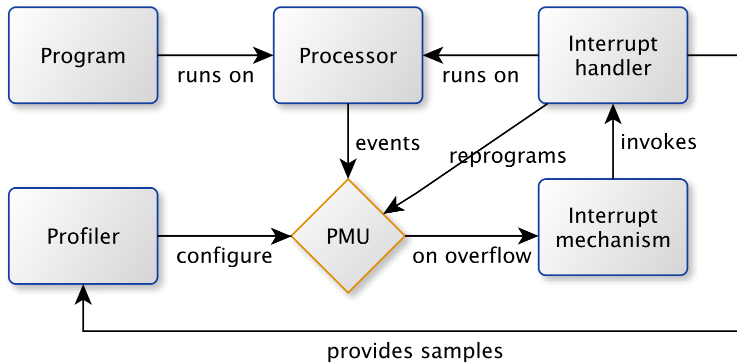



**Figure 3.1:** Sampling process flowchart

5. The `Perfmon` context is actually created by using the `pfm_create_context` command.

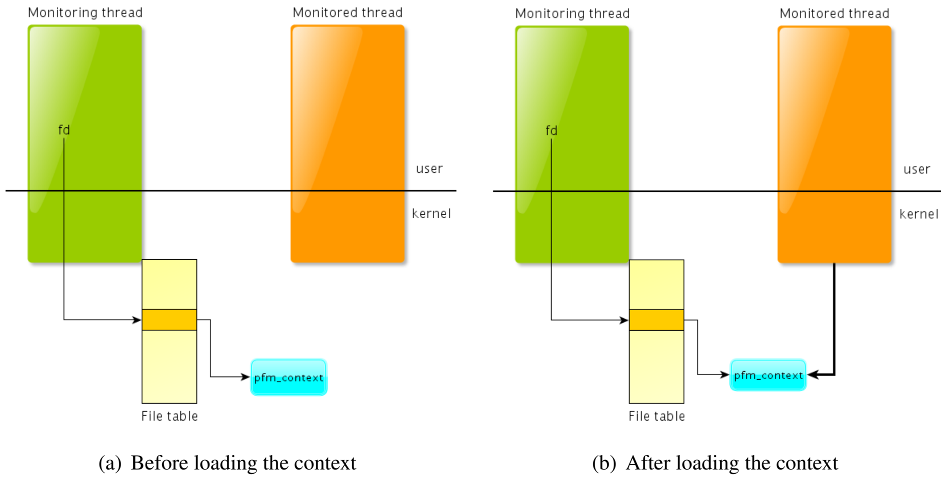
6. The context is attached to the pid/tid of the process/thread to be monitored by using the `pfm_load_context` command.

7. Monitoring is started and, eventually, stopped, by using the `pfm_start` and `pfm_stop` commands, respectively.

There are two points from the previous enumeration to highlight: the way a context is attached to a thread, and how the sampling buffer is accessed. The effects of `pfm_load_context` are shown in Figure 3.2. The thread to which the context is attached is called *monitored thread*. The thread which sets up `Perfmon` and monitors the *monitored thread* is called *monitoring* or *controlling thread*. A thread can monitor itself (*monitoring thread* = *monitored thread*) and therefore, it is called *self-monitoring thread*. The context is accessed by the monitoring thread through a file descriptor (`fd`) in the file table.

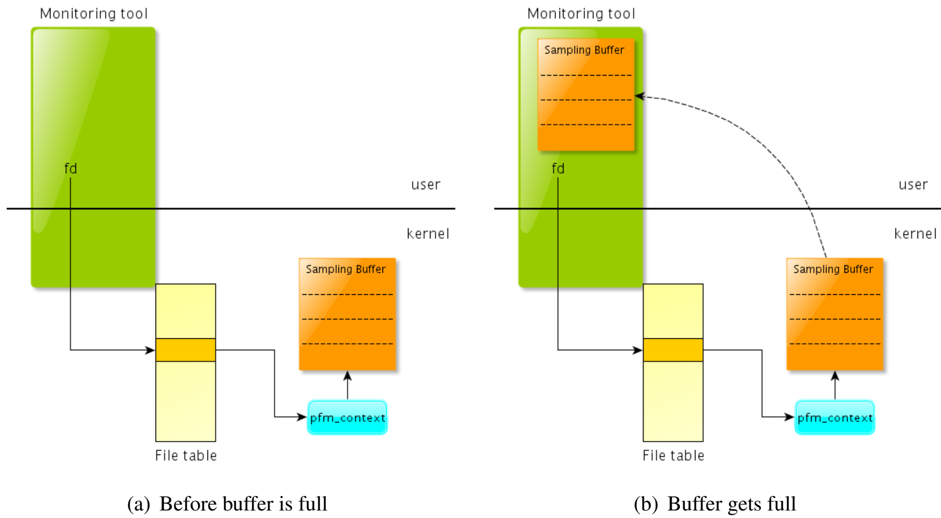
Regarding the sampling process, when a PMU is monitoring a thread and the counter which counts the number of captured EAR events overflows (it will do it sooner or later depending on the sampling period and the number of events generated by the code), the data from PMDs 32, 33 and 36[1] are stored as an entry in the kernel sampling buffer. This goes on

[1] On Itanium2 Montvale.




(a) Before loading the context (b) After loading the context

**Figure 3.2:** Effect of attaching a context to a monitored thread.




(a) Before buffer is full (b) Buffer gets full

**Figure 3.3:** The sampling buffer is mapped to user level when it gets full.

until the buffer is full –when all the collected entries reach a whole size as large as the buffer size–. Then, a notification is sent and the sampling buffer is mapped into user level, which makes possible for the controlling thread to read all the values stored in the buffer using the notification signal handler. This behaviour is depicted before a buffer interrupt occurs (Figure 3.3(a)) and after it (Figure 3.3(b)).

Algorithm 1 shows a C-based algorithm of a self-monitoring, OpenMP SpMV code using `Perfmon` and `libpfm` to capture EAR events on a Montvale processor. This code or any variation of it will be used through the rest of this dissertation.

The code shows in the first place a `process_smpl_buffer` and an `overflow_handler` methods. The former reads the sampled values of the `EVENT_NAME` event from the sampling buffer when it gets available to user level. The latter invokes the former when an interruption is delivered after the buffer overflows. Then, when `process_smpl_buffer` returns, the overflow handler informs `Perfmon` that the notification processing is finished, and restarts the monitoring process[2]. Once inside the `main` method, some initial tasks are performed (lines 20 to 22): `libpfm` is initialised, the overflow handler is configured to invoke the `overflow_handler` method and a search to locate the event to monitor is carried out. Next, all the subsequent actions are performed for all the threads of the parallel region created by "`pragma omp parallel`". `libpfm` is used to find out which PMC and PMD registers must be programmed depending on the event and the processor (line 29). The next step consists in creating a new `Perfmon` context for each thread (line 32) by invoking the `perfmonctl()` function, key piece of the `Perfmon` interface. As shown in Figure 3.2, contexts are accessed through a common file descriptor for each thread (`fd[tid]`). After configuring the sampling buffer, sampling period and other information such as the event to monitor in the related PMCs and PMDs (lines 38 to 42), each context is attached to the monitoring task through `Perfmon` (line 45). Finally, monitoring is started and stopped using the `libpfm` functions `pfm_self_start` and `pfm_self_stop` (lines 51 and 60, respectively).

## 3.3 Evaluation of Perfmon on FINISTERRAE

In this section two cases of study are included to assess the use of `Perfmon` on the FINISTERRAE architecture. The first one is the sparse matrix-vector product. The second one is a dense regular loop. The overhead imposed by the use of `Perfmon` is also characterised.

[2] After a buffer overflow, monitoring is disabled and must be manually restarted.

**Algorithm 1:** self-monitoring SpMV algorithm using `Perfmon`.

```
1:  #include < perfmon/perfmon.h >
2:  #include < perfmon/perfmon_default_smpl.h >
3:  #include < perfmon/pfmlib_montecito.h >
4:  #define EVENT_NAME "data_ear_cache_lat4"
5:
6:  overflow_handler(){
7:  process_smpl_buffer()
8:  perfmonctl(PFM_RESTART, tid)
9:  }
10:
11: process_smpl_buffer(){
12: count ← samples in smplBuffer[tid]
13: while (count − −) do
14:     [addr, latency] ← getSample(count)
15:     print addr, latency
16: end while
17: }
18:
19: main(){
20: initialize_pfm_library()
21: install_overflowHandler()
22: pfm_find_full_event(EVENT_NAME)
23:
24: #pragma omp parallel
25: {
26: tid ← omp_get_num_thread()
27:
28: /* let the library figure out the values for the PMCS */
29: pfm_dispatch_events()
30:
31: /* create the context for self monitoring */
32: ctx[tid] ← perfmonctl(PFM_CREATE_CONTEXT)
33:
34: /* extract file descriptor for our context */
35: fd[tid] ← ctx[tid].fd
36:
37: /* configure sampling */
38: smplBuffer[tid] ← ctx[tid].smplBuffer
39: pd[tid] ← sampling period
40: pc[tid] ← sampling config flags
41: perfmonctl(fd[tid], PFM_WRITE_PMCS, pc)
42: perfmonctl(fd[tid], PFM_WRITE_PMDS, pd)
43:
44: /* attach context to our task */
45: perfmonctl(PFM_LOAD_CONTEXT, fd[tid])
46:
47: /* setup asynchronous notification on the file descriptor */
48: fcntl(fd[tid], F_SETFL)
49:
50: /* start monitoring */
51: pfm_self_start(fd[tid])
52: #pragma omp for
53: for (i = 0; i < N; i + +) do
54:     Y[i] ← 0.0
55:     for (j = A.ptr[i]; j < A.ptr[i + 1]; j + +) do
56:         Y[i] ← Y[i] + A.value[j] ∗ X[A.index[j]]
57:     end for
58: end for
59: /* stop monitoring */
60: pfm_self_stop(fd[tid])
61: }
62: end main
```

### 3.3.1 Sparse Matrix-Vector product as a case of study

Perfmon was used to validate the use of EAR sampling for a version of the sparse matrix-vector product. The code used is the one shown above in Algorithm 1. The sparse matrix $A$ is stored in CRS format [58]. The three components of $A$ are $ptr$, $index$ and $value$, which store the index of the first entry in each row, the column index and the value of the entry, respectively. $X$ is the vector to be multiplied by $A$, and $Y$ is the vector that stores the final result of the operation. Note that the indirection is in vector $X$, and it is irregularly accessed according to $index$. The main objective of this experiment has been to study the behaviour of the accesses to $X$. To do that, only accesses to vector $X$ were considered by applying data triggering at specific addresses, functionality which is provided by Perfmon.

Note that, although not showed in Algorithm 1, the matrix is read by the main thread before executing the SpMV, being subsequently touched by each thread. Therefore, data are very likely to be found in the cache memory of each core. In this regard, the expected average access latency to data will be in the range of 5 to 14 cycles, which corresponds to accesses to L2 or L3 cache memory [2].

The chosen EAR event to capture was DATA_EAR_CACHE_LAT4, which makes Perfmon capture L1 cache misses with latencies higher than 4 cycles. As the L1 is an integer-only cache in the Montvale family, the highest level in the cache hierarchy where all accesses to float or double values are stored is the L2 cache. These accesses are always considered as L1 misses by the Montvale's PMU. Therefore, all those accesses are susceptible of being sampled. Note that the sampling period was set to 10, which means that one out of each ten L1 cache misses will be sampled. This period is low enough to capture a set of samples representative of the original signal[3] [75].

To perform this experiment, a set of 24 matrices was selected from the matrix test set presented in Table 1.1. Their patterns are shown in Figure 3.4. Tests were performed in a stand-alone node of the FinisTerrae. Cases from 2 to 16 threads were considered. Table 3.1 shows the characteristics of the matrix set and the percentage of entries that were actually sampled by the EARs. The sampled percentage is the ratio between the number of detected accesses and all the possible accesses.

Results show that the average percentage of sampled entries for this code is always under 20% and approximately constant. In this regard, it can be concluded that the amount of

[3] An in-depth study of the sampling representativeness is performed in Chapter 5.

| | Matrix | Size | Entries | *Sampled entries* (%) | | | |
|---|---|---|---|---|---|---|---|
| | | | | $2th$ | $4th$ | $8th$ | $16th$ |
| $a$ | $bcsstk20$ | 485 x 485 | 1810 | 15 | 15 | 15 | 16 |
| $b$ | $bcsstk05$ | 153 x 153 | 1288 | 20 | 20 | 19 | 19 |
| $c$ | $bfw398b$ | 398 x 398 | 2910 | 10 | 11 | 10 | 10 |
| $d$ | $bcsstk04$ | 132 x 132 | 1890 | 18 | 18 | 17 | 18 |
| $e$ | $lnsp131$ | 131 x 131 | 536 | 16 | 15 | 15 | 15 |
| $f$ | $mhd416a$ | 416 x 416 | 8562 | 19 | 19 | 19 | 19 |
| $g$ | $west0381$ | 381 x 381 | 2157 | 18 | 18 | 18 | 18 |
| $h$ | $mbeaflw$ | 496 x 496 | 49920 | 12 | 14 | 14 | 12 |
| $i$ | $mhd4800b$ | 4800 x 4800 | 27520 | 15 | 15 | 15 | 15 |
| $j$ | $bcsstk28$ | 4410 x 4410 | 111717 | 19 | 19 | 19 | 19 |
| $k$ | $bcsstk23$ | 3134 x 3134 | 24156 | 17 | 17 | 18 | 18 |
| $l$ | $bcsstk24$ | 3562 x 3562 | 81736 | 18 | 18 | 18 | 18 |
| $m$ | $lnsp3937$ | 3937 x 3937 | 25407 | 16 | 16 | 16 | 16 |
| $n$ | $sherman2$ | 1080 x 1080 | 23094 | 15 | 15 | 16 | 15 |
| $o$ | $west2021$ | 2021 x 2021 | 7353 | 18 | 18 | 19 | 18 |
| $p$ | $psmigr_1$ | 3140 x 3140 | 543162 | 12 | 12 | 12 | 12 |
| $q$ | $bcsstk25$ | 15439 x 15439 | 133840 | 18 | 18 | 18 | 18 |
| $r$ | $e40r0000$ | 17281 x 17281 | 553956 | 9 | 9 | 9 | 9 |
| $s$ | $bcsstk18$ | 11948 x 11948 | 80519 | 17 | 17 | 17 | 17 |
| $t$ | $bcsstk17$ | 10974 x 10974 | 219812 | 19 | 19 | 19 | 19 |
| $u$ | $fidapm29$ | 13668 x 13668 | 186294 | 18 | 18 | 18 | 18 |
| $v$ | $fidap019$ | 12005 x 12005 | 259863 | 19 | 19 | 19 | 19 |
| $w$ | $af23560$ | 23560 x 23560 | 484256 | 17 | 17 | 17 | 17 |
| $x$ | $memplus$ | 17758 x 17758 | 126150 | 18 | 18 | 18 | 20 |

**Table 3.1:** List of the matrix test set and percentage of sampled entries by `Perfmon` for 2, 4, 8 and 16 threads.

sampled data is not influenced by the pattern, type, size or number of non-zero entries of the input matrix. Nor is it by the number of threads.

### 3.3.2 Accesses to a dense vector as a case of study

A different kernel was used to characterise the use of `Perfmon` with regular codes. The code is shown in Algorithm 2.

In this case, vectors $X$ and $Y$ are accessed in a regular way. Loop $j$ is used to artificially increase the computation time in each iteration of loop $i$ in a controlled way (by tuning values of M). Variables $res$, $alpha$ and $beta$ are local to each thread, whereas $X$ and $Y$ are global. Accesses to Vector $X$ are sampled by `Perfmon`. These vectors are read by the main thread

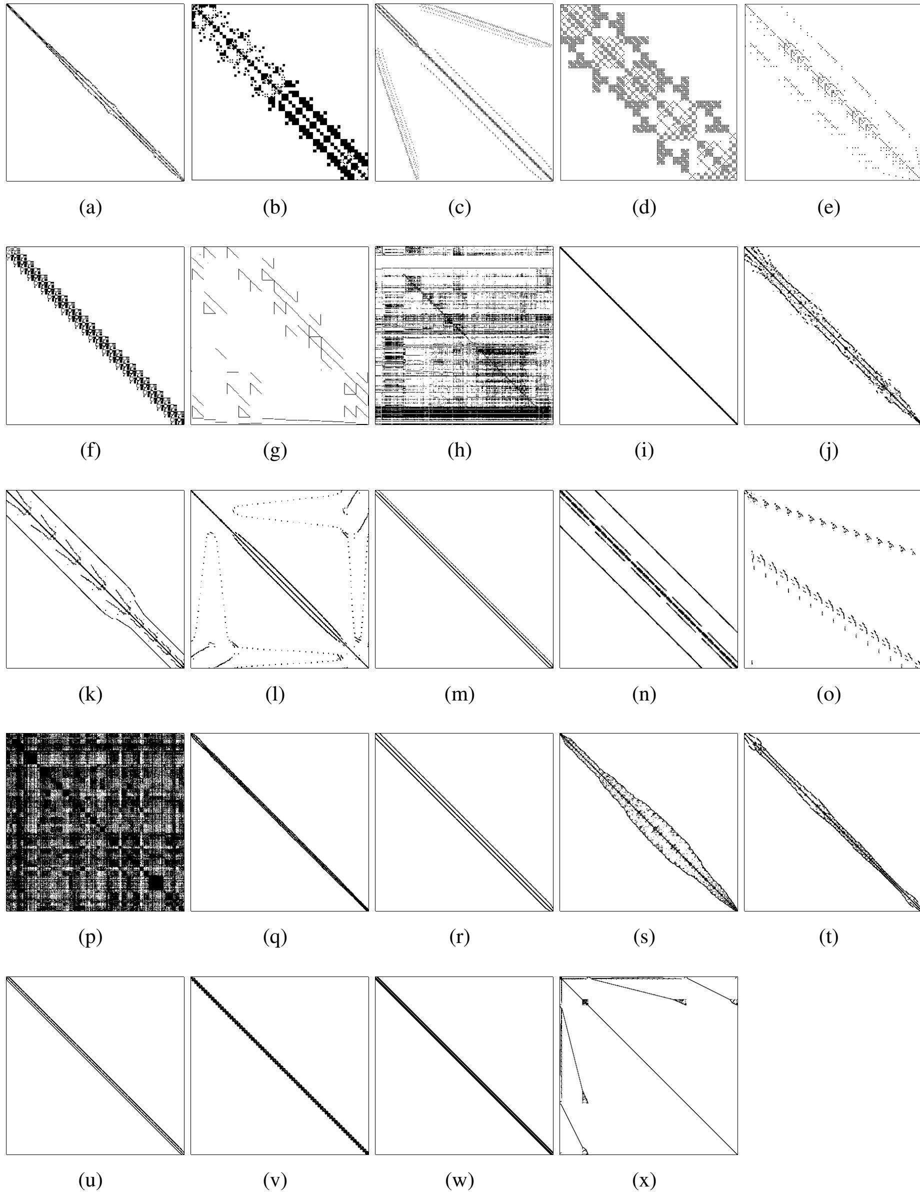



**Figure 3.4:** Set of matrices used to study the SpMV: a) `bcsstk20`, b) `bcsstk05`, c) `bfw398b`, d) `bcsstk04`, e) `lnsp 131`, f) `mhd416a`, g) `west0381`, h) `mbeaflw`, i) `mhd4800b`, j) `bcsstk28`, k) `bcsstk23`, l) `bcsstk24`, m) `lnsp3937`, n) `sherman2`, o) `west2021`, p) `psmigr1`, q) `bcsstk25`, r) `e40r0000`, s) `bcsstk18`, t) `bcsstk17`, u) `fidapm29`, v) `fidap019`, w) `af23560` and x) `memplus`.

**Algorithm 2:** Dense algorithm to test `Perfmon`.

```
1:  #pragma omp parallel
2:  {
3:  #pragma omp for
4:  for (i = 0; i < N; i++) do
5:      Y[i] ← alpha × X[i]
6:      for (j = 0; j < M; j++) do
7:          res ← res × beta
8:      end for
9:  end for
10: }
```

before executing the kernel, being subsequently touched by each thread. Therefore, data are very likely to be found in the cache memories of each core. Note that, because of the regularity of the code, some mechanisms like prefetching and cache line reuse will exercise an important influence on the performance.

This code was run considering different values of $N$, $M$, number of threads and different types of data for vectors $X$ and $Y$ (integers, floats and doubles). Figure 3.5 shows the results for integers and floats. Results for doubles are similar to those shown for floats.

Some observations can be made from these results. Figure 3.5(a) shows that the number of entries that are detected by the EARs is about 11%. Note that this result is consistent with the ones shown in the previous section using an irregular kernel. They also seem to be independent of the number of threads and to slightly decrease as the size of the vector increases.

Figure 3.5(b) shows that the amount of work executed in the loop increases with the number of entries detected by the EARs. Figure 3.5(c) shows the same behaviour as Figure 3.5(b) but for a vector of integers instead of floats. In this case, the influence of $M$ is more important. Note that the percentage of sampled entries is much lower mainly because many integers can be in the L1 cache, therefore not being captured by the EARs. Finally, Figure 3.5(d) shows the effect of the sampling period used by `Perfmon`. Note that as this period doubles, the number of sampled entries is approximately reduced by half, as expected. This behaviour is broadly independent of the vector size.

### 3.3.3 Evaluation of `Perfmon` and `libpfm` overhead

A series of tests was performed to quantify the overhead introduced by the inclusion of the `Perfmon` and `libpfm` calls and how much can affect the performance of a monitored program. This section explains the methodology followed and presents the results obtained.

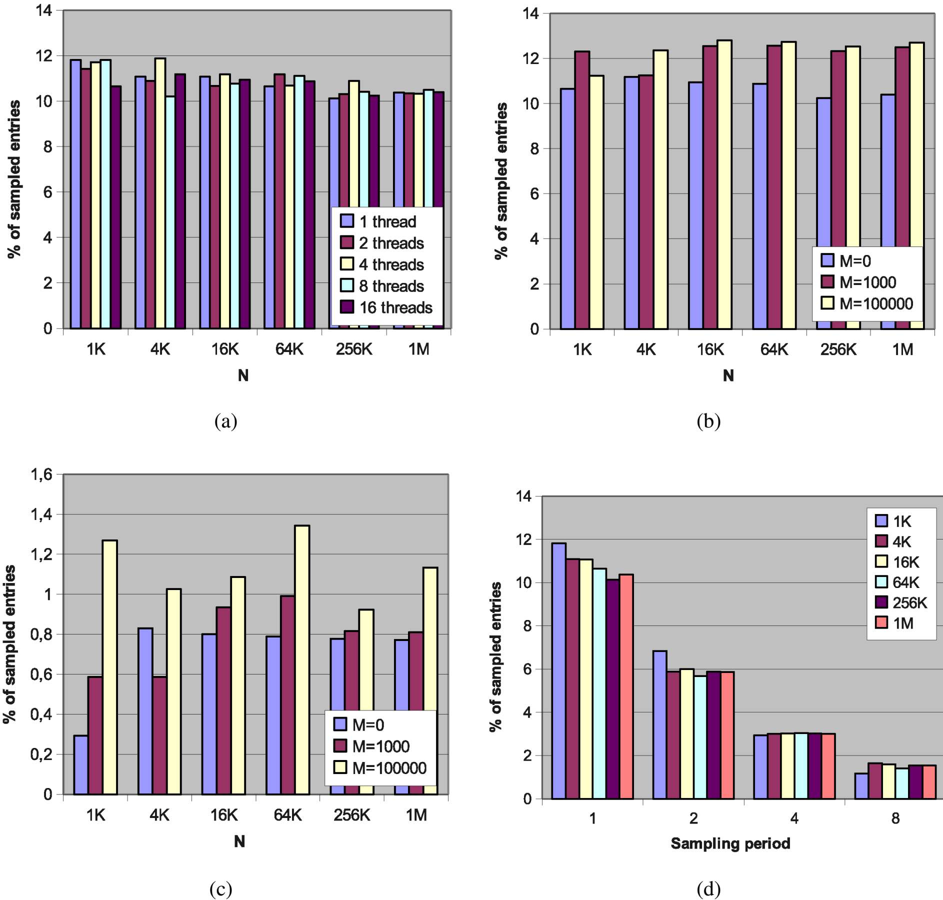



**Figure 3.5:** Ratio between the number of sampled entries of vector $X$ and its size. **(a)**: for different values of $N$ and number of threads; $X$ is a vector of floats. **(b)**: for different values of $N$ and $M$; $X$ is a vector of floats. **(c)**: for different values of $N$ and $M$; $X$ is a vector of integers. **(d)**: for different values of $N$ and the sampling period; $X$ is a vector of floats.

The self-monitored, parallel SpMV of Algorithm 1 was used as a benchmark. A typical session with `Perfmon` and `libpfm` was considered, analysing their most usual calls. The routines studied are:

- pfm_set_options
- pfm_get_pmu_type
- pfm_find_full_event
- pfm_dispatch_events
- pfm_create_context
- pfm_write_pmcs
- pfm_write_pmds
- pfm_load_context
- pfm_self_start
- pfm_restart
- pfm_self_stop
- pfm_initialize

The execution time of each call was measured. For each matrix of Figure 3.4 those routines were invoked, executing the code 100 times per matrix.

Figure 3.6 shows the percentage of time consumed by each function normalised to the average execution time of the whole matrix test set. Note the following facts:

- The most time-consuming functions are always `pfm_find_full_event`, `pfm_dispatch_events` and `pfm_create_context`. The first one is a general-purpose search routine which, given an event name, returns an event descriptor. `Pfm_dispatch_events` sets up the values to program the PMC registers. Finally, `pfm_create_context` creates the context for self-monitoring. These are functions that are invoked just once during the execution of the whole program.

- In theory, the functions susceptible of introducing more overhead are `pfm_self_start` and `pfm_self_stop` –since they could be invoked several times in a code– and `pfm_self_restart`, which will be invoked after every buffer overflow. The number of buffer overflows occurred during a program execution depends on the sampling period and the number of sampled events, so its overhead is not straightforward to estimate. A rough estimation, based on an average number of 700 overflows measured in our observations, gives an `pfm_self_restart` overhead of 0.1%.

- All the remaining overheads are under 0.01% for the whole matrix set considered.

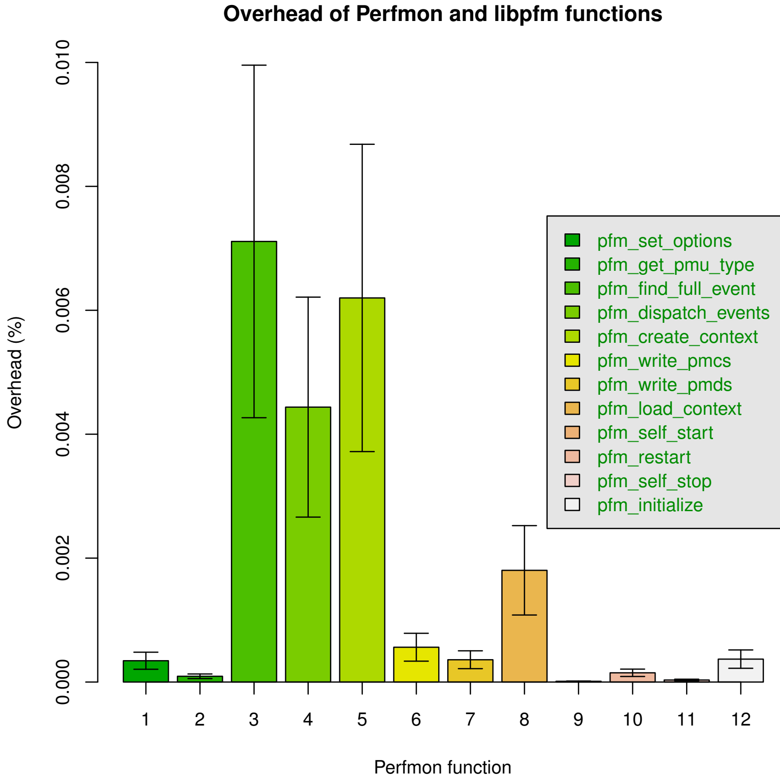



**Figure 3.6:** Percentage of time consumed by the `Perfmon` and `libpfm` functions on a SpMV program.

To summarise, both `libpfm` and `Perfmon` are suitable to be used in our measurements, since the introduced overhead is affordable and can be amortised with the improvement obtained in the developed techniques.

## 3.4 Conclusions

This chapter has addressed a study of the set of features provided by the hardware counters in the Itanium2 platform (goal G2-HwCStudy). Firstly, the choice of tools used in this dissertation to access the Itanium2 PMU (the `Perfmon` interface and its higher-level library `libpfm`) has been justified. Next, the internals of `Perfmon` and the procedure to program

hardware counters has been explained. An example code has been then developed, in order to serve as a future reference for this dissertation.

In this chapter, `Perfmon` has been evaluated on our test infrastructure, the FINISTERRAE supercomputer. Both dense and sparse codes have been used. Results show that the percentage of sampled information is independent of the monitored load. Finally, additional experiments have proved the overhead of the interface to be negligible.

The next chapter will rely on this study to seek methods of improving the locality of irregular codes using hardware counters (G3-REORDTECHIMPRV).

# 4

# *Locality Improvement on Irregular Codes*

## 4.1 Introduction

Numerous methods have been proposed in the literature to optimise the locality of irregular codes in parallel architectures. A particular case is the family of techniques that modify the allocation of data structures in memory. These techniques typically comprise an initial stage in which the input data must be analysed and reordered to achieve enough locality. The computational cost of this stage is dependent on the input data size and not until several iterations of a computational kernel does its execution compensate the time spent in the reordering stage. This type of heuristics is particularly useful in applications based on iterative methods.

In this line, an optimisation technique was developed in our research group and presented in a previous dissertation [76]. This technique reorders the data structures of the irregular code considered (sparse matrices) using heuristic techniques guided by a locality model that is general enough so that it can be used in numerous codes of sparse matrix algebra. It is based on the following premise: *the access locality is characterised by a sparse matrix pattern, so the closer the entries are to each other in the pattern, the higher the locality of the accesses addressed by the matrix will be*. Like all models from this family, the computational cost of this technique is typically high as it considers all nonzeros of the sparse matrix in order to find an appropriate permutation of rows and columns from the input matrix.

As introduced in Chapter 1, the sparse matrix-vector product (SpMV) is representative of the paradigm of irregular codes with low data reuse caused by irregular and indirect memory access patterns. Hence, this kernel is a good candidate to test the optimisation technique mentioned above to meet one of the goals proposed in this dissertation (G3-REORDTECHIMPRV): to develop models that make use of the information provided by hardware counters to improve the locality of irregular codes.

This chapter studies the feasibility of increasing the locality of the SpMV when only a subset of the memory accesses performed is available in the optimisation process, which is equivalent to consider only a subset of the nonzero elements of the matrix. The ultimate objective is to use incomplete information to reduce the cost of the reordering stage in the locality method. Two approaches are proposed to obtain such a nonzero subset. On the one hand, a common random sampling method. On the other hand, a novel method to characterise matrices using the information provided by the hardware counters. Subsequent sections will prove how the use of incomplete information can achieve a dramatic time reduction without performance loss and, particularly, how hardware counters can provide such incomplete information in a reliable and costless way.

The remainder of this chapter is structured as follows: for the sake of clarity, Section 4.2 introduces the locality optimisation technique previously developed in our research group. Section 4.3 evaluates this technique in a SpMV with randomly sampled matrices. Section 4.4 introduces a novel sampling method using hardware counters and compares it with the original locality optimisation technique. An evolution of this method using additional information from the hardware counters is presented in Section 4.5. Finally, the chapter concludes with some remarks and future work in Section 4.6.

## 4.2 Locality optimisation technique

The data reordering technique introduced in [12] has been used to study the locality optimisation of a matrix considering only a subset of the memory accesses performed by the SpMV code. This technique reorganises the data guided by a locality model instead of restructuring the code or changing the sparse matrix storage format. Unlike other existing locality models that predict the data movement along the levels of the memory hierarchy, this model is able to characterise, sacrificing accuracy, the trend of this movement in general terms. In particular, locality is evaluated using a distance function that depends on the number of entry matches ($a_{elems}$). Considering accesses to the sparse matrix by rows, the number of entry matches between any pair of rows is defined as the number of nonzero elements in the same column of both rows. In this way, the distance between rows $i$ and $j$ is defined as:

$$d(i,j) = n_{\mathrm{elems}}(i) + n_{\mathrm{elems}}(j) - 2 * a_{\mathrm{elems}}(i,j) \qquad (4.1)$$

where $n_{elems}(i)$ is the number of entries in row $i$. This function is a norm, and it is used to measure the locality displayed by the accesses performed by the SpMV code on these two

rows when they are consecutively accessed. For a given sparse matrix accessed by rows, a measure that is inversely proportional to the data locality for the whole sparse matrix can be defined as follows:

$$D = \sum_{i=0}^{n-2} d(i, i+1) \tag{4.2}$$

where $n$ is the number of rows/columns of the sparse matrix. These definitions can directly be extended to columns. Note that these functions only provide results based on the locality evaluated on pairs of consecutive rows (or columns) of the sparse matrix. Nevertheless, reuse of data could be possible at any level of the memory hierarchy during the product of two or more consecutive rows (or columns) of the matrix. For this reason, a generalisation of the distance functions based on the concept of *windows of locality* can also be defined.

A window of locality is a set of $w$ consecutive rows (or columns) of the matrix between which there is a high probability of data reuse when executing the sparse matrix code. Based on the distance function $d(i, j)$, we can define the distance between windows of locality $g$ and $h$ as:

$$d_w(g,h) = n(g) + n(h) - 2 * a_{\text{elems}}(g,h) \tag{4.3}$$

where $n(g) = n_{elemsw}(g) - a_{elemsw}(g)$. The parameter $a_{elems}(g,h)$ is a direct extension of the entry matches between windows $g$ and $h$. $n_{elemsw}(g)$ is the number of elements of window $g$, and $a_{elemsw}(g)$ generalises the concept of entry matches considering those that take place on two or more rows within window $g$. Note that, by introducing $n(g)$, the possible reuse of data inside $g$ is also considered. Figure 4.1 shows an example of the calculation of these parameters when $w = 2$. Therefore, the indirect estimation of locality defined for a sparse matrix in Equation 4.2 can now be calculated as a sum over the whole matrix:

$$D_w = \sum_{g} d_w(g, g+1), \quad \forall\, g \mid 0 \leq g < \lceil n/w \rceil \tag{4.4}$$

Note that these distances (Equations 4.3 and 4.4) are equivalent to the distances measured over pairs of consecutive rows/columns of the matrix when the window size is $w = 1$.

The reordering technique modifies the pattern of the sparse matrix according to the locality model described before. In order to increase the locality in the accesses performed by the SpMV for a given matrix, a permutation of windows of locality that minimises its total distance $D_w$ must be found (Equation 4.4). The problem of locality improvement is formulated

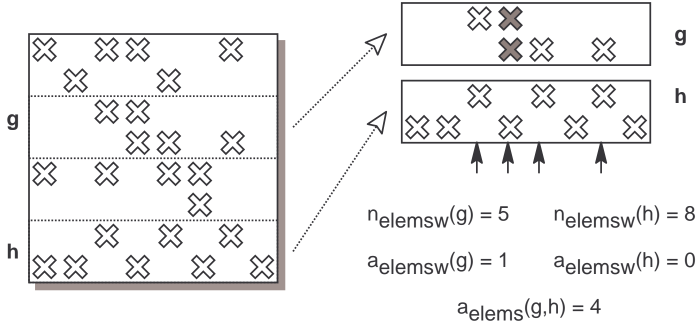



**Figure 4.1:** Calculation example of $n_{\text{elemsw}}(g)$, $a_{\text{elemsw}}(g)$ and $a_{\text{elems}}(g,h)$.

as a classic NP-complete optimisation problem, and it is solved as a graph problem using its analogy to the *traveling salesman problem* (TSP) [77].

The problem is described using a weighted graph where each node represents a window of locality of the input sparse matrix. Each edge of the graph has an associated weight that reflects the distance between pairs of windows of locality according to the description of locality given previously. Nevertheless, it is unnecessary to work with a complete graph. Given that sparse matrices have a very low density of nonzero elements, most of the weights in the graph correspond to cases where $\text{a}_{\text{elems}} = 0$. Those values, according to the distance definitions, represent the worst locality cases. So, without losing relevant information about locality, we can use an incomplete weighted graph where only values of $\text{a}_{\text{elems}}$ different from zero are considered. As a consequence, the graph size is noticeably reduced.

Solving the reordering problem is equivalent to finding a path of minimum length that goes through all the nodes of the graph. This path is represented as a *permutation vector* that appropriately sorts the nodes of the graph, giving a reordered matrix. To find the best path, a heuristic solution was chosen, given that the distance measures defined previously can be used to validate the quality of a rearragement. After a comparative study of different techniques, the *Chained Lin–Kernighan* heuristic proposed by Applegate et al. [78] was chosen. Note that, for practical purposes, if there is an isolated node without edges in the distance graph, the TSP heuristic will internally connect this node to the others through edges with a very high distance.

In order to select the window size ($w$), two types of windows of locality are considered: fixed and variable [12]. For windows of fixed size the number of nodes in the weighted graph is $\lceil n/w \rceil$. Therefore, for high values of $w$, the graph is noticeably reduced. It implies

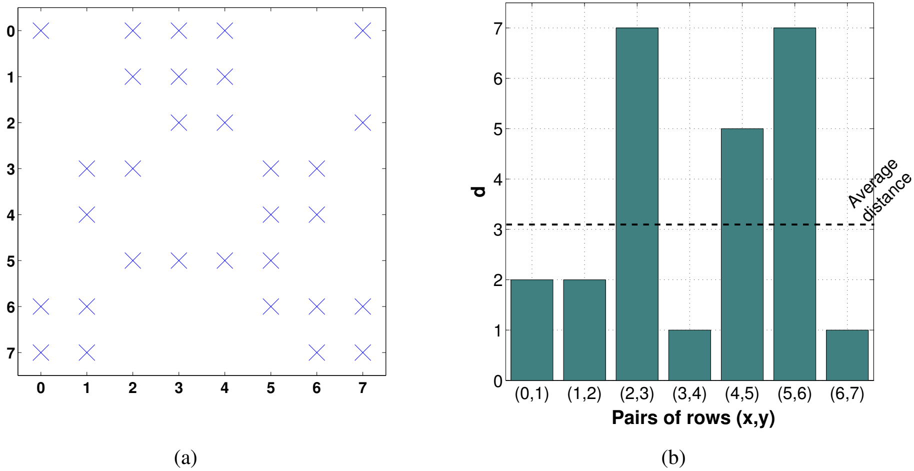



**Figure 4.2:** Example of windows of locality with variable size: (a) sparse matrix example. (b) distance histogram.

an important decrease in the computational time needed for its calculation, together with a reduction in the size of the problem to be managed by the reordering heuristic. Note that we are not taking into account any locality property of the input sparse matrix in order to define the windows. Therefore, windows of locality can comprise consecutive rows (or columns) of the matrix which exhibit low locality according to our model.

There is a two-fold purpose in using windows of locality of variable size. On the one hand, as in the fixed size case, windows are used to decrease the number of nodes in the weighted graph. On the other hand, they avoid grouping consecutive rows (or columns) of the matrix with low locality in the window creation process. Figure 4.2 shows an example of the technique to create windows of variable size considering the rows of the matrix. First, a histogram is created from the input matrix. It represents the distance between each pair of consecutive rows. Therefore, there are $n-1$ values in the histogram. In order to decide if two consecutive rows $i$ and $j$ will be included within the same window a simple criterion must be fulfilled: $d(i,j) < D/n$. That is, the distance must be lower than the average distance of the whole sparse matrix. According to this, in the example of Figure 4.2, four windows of locality are defined: $\{0,1,2\}$, $\{3,4\}$, $\{5\}$ and $\{6,7\}$. This way, the windows creation process is guided by the locality model and, as a consequence, locality among rows within each window is increased. This process can directly be extended to columns. Note that when the matrix

| *Matrix* | # *rows* ($n$) | # *nonzeros* ($nnz$) | $nnz/row$ |
|---|---|---|---|
| $crystk03$ | 24696 | 1751178 | 71 |
| $garon2$ | 13535 | 390607 | 29 |
| $gyro_k$ | 17361 | 1021159 | 59 |
| $mixtank_new$ | 29957 | 1995041 | 67 |
| $msc10848$ | 10848 | 1229778 | 113 |
| $nd3k$ | 9000 | 3279690 | 364 |
| $nmos3$ | 18588 | 386594 | 21 |
| $pct20stif$ | 52329 | 2698463 | 52 |
| $sme3Da$ | 12504 | 874887 | 70 |
| $tsyl201$ | 20685 | 2454957 | 119 |

**Table 4.1:** Matrix benchmark suite used in our sampling tests.

pattern is non-symmetric, the windows creation process must be applied considering rows and columns independently.

In the studies performed in [12] it is concluded that windows of $w = 1$ and $w = variable$ are the best choices in terms of performance. For this reason, this dissertation focuses on them. However, big fixed-sized windows can be a good solution due to the particularities of the sparse matrices in some applications such as those related to the simulation of semiconductor devices [79].

## 4.3 Locality optimisation using randomly sampled matrices

This section studies the behaviour of the reordering technique introduced above to estimate the locality of a given matrix by using just a percentage of its nonzero elements. This study intends to check whether reordering techniques can be used with incomplete information and to state the amount of information required to obtain similar results to the original reorderings. For that purpose, a subset of the general matrix set presented in Chapter 1, shown in Table 4.1, was chosen. Then, a set of sampled matrices from the original ones was generated. In particular, the sampled matrices contain 1%, 2%, 5%, 10%, 15% or 20% of the original nonzeros. These nonzeros are randomly selected using the C `random()` function, which implies that each nonzero of the original matrix has equal chances of being sampled. Twenty sampled matrices were generated for each matrix and percentage, which means a total amount of 1200 sampled matrices to test. Figure 4.3 shows an example of the nonzeros pattern of some of the randomly sampled matrices generated from the `nmos3` matrix.

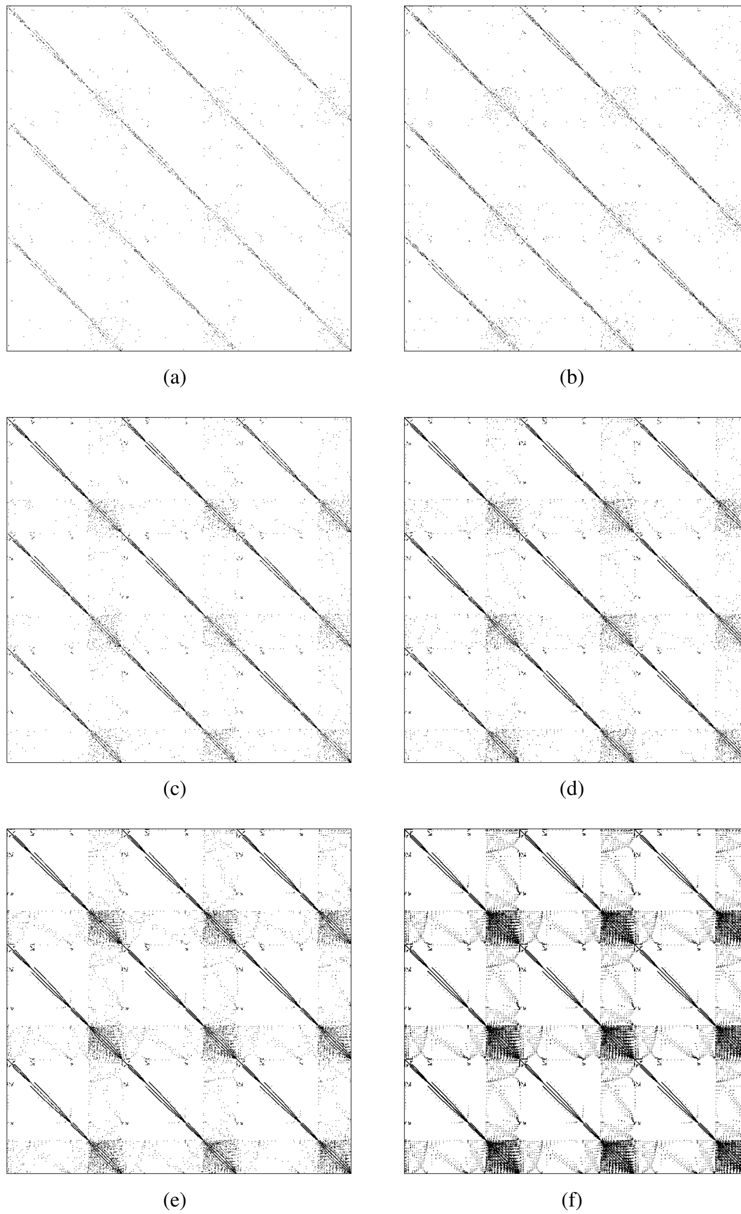



**Figure 4.3:** Examples of `nmos3` randomly sampled matrices. Matrices with 1% (a), 2% (b), 5% (c), 10% (d) and 20% (e) of the nonzeros with respect to the original matrix (f).

The information provided by the sampled matrices has been used to generate a permutation vector that will be applied to the original matrix. This permutation vector is calculated by the reordering technique using the sampled matrices with windows of both variable and fixed ($w = 1$) size (see Section 4.2). In this way, the original matrix is reordered considering only the information provided by a subset of its nonzeros.

In order to estimate the quality of these reorderings, a comparison with the original technique (that is, when the not-sampled matrix is used to calculate the permutation vector) has been carried out. Figures 4.4 and 4.5 show the normalised SpMV performance (a value of 1 means the same result as the original technique) for the FINISTERRAE system. These figures display the average, the maximum and the minimum performance for each sampled matrix set and number of threads. Note that, due to the random sampling process, the twenty sampled matrices of each subset can be very different. Given that these sampled matrices are used as an input of the reordering technique, the output in each case may show a high variation, which will cause fluctuations in the observed performance within each subset.

Focusing on the results using a fixed-size window reordering ($w = 1$), shown in Figure 4.4, it is noticeable that reorderings using sampled and complete information show a similar behaviour. This happens even in those cases in which only 1% of the nonzeros are considered, which means that the locality model used by the reordering technique is able to characterise the accesses performed by the sparse matrix using only this information. There are also some cases where the sampled nonzeros are not enough to compete with the original technique. Take for instance the matrices `gyro_k` and `sme3Da` with two threads, or `mixtank_new` with four threads. However, the difference with the reference is, at most, about 5%. Conversely, there are some cases where reorderings using sampled information outperform the original ones. For example, reorderings of `nd3k` and `pct20stif` with eight threads.

The explanation for this behaviour can be found in some details of the locality model and the reordering heuristic. The reordering technique uses the matrix as the input to create the distance graph (see Section 4.2). Therefore, the sampled matrices generate graphs that are different from the one produced by the original matrix. Since the sampled matrices have fewer nonzeros than the original ones, the number of entry matches ($a_{elems}$) will be smaller, so the graph will be reduced, displaying a smaller number of edges. Therefore, given that the graphs are different and the chosen strategy to minimise the total distance is heuristic, the results will be different. Note that the TSP heuristic is constrained to a fixed number of iterations searching for equilibrium between performance and overhead. Hence, there will be

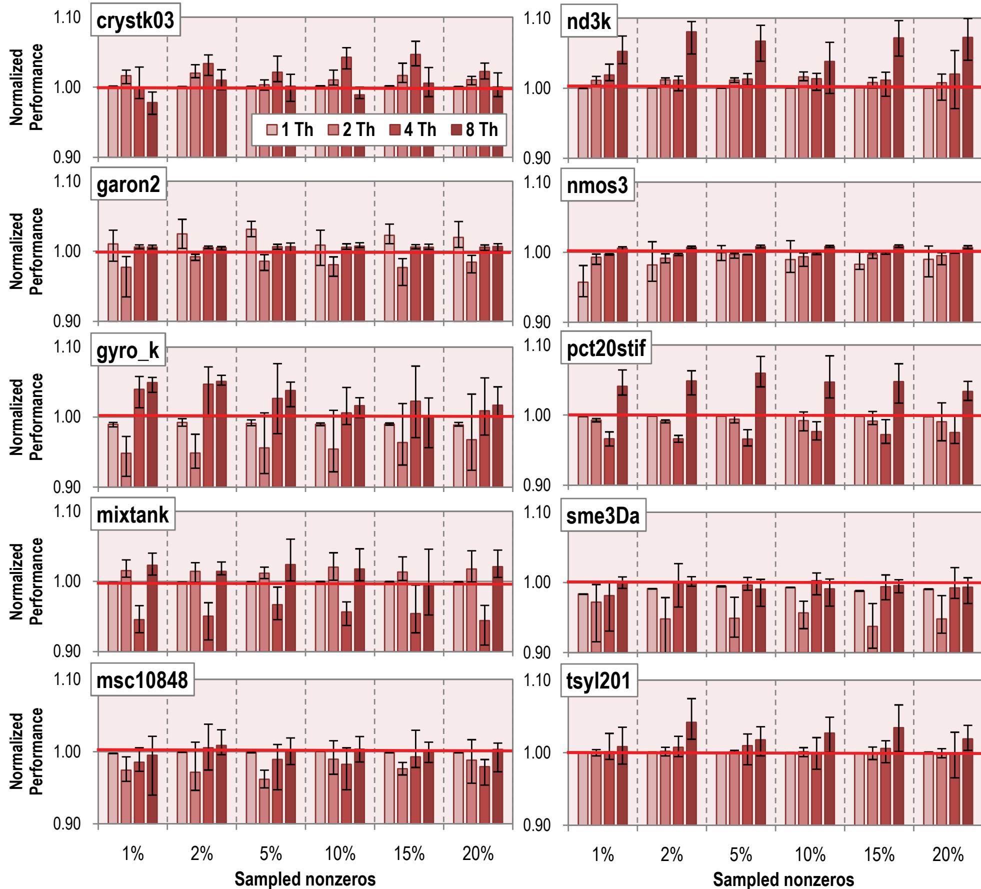



**Figure 4.4:** Normalised SpMV performance obtained by the reorderings generated using the locality optimisation technique ($w = 1$) and the information provided by the randomly sampled matrices on the Itanium2 platform.

cases where the reordering heuristic, using a "sampled graph", will produce a better solution than the original one, because the problem considered is smaller.

Another observation is that performance results change depending on the considered number of threads. Note that the reordered matrices are generated without taking into account the number of threads used to perform the SpMV. However, the sparse matrix is distributed among the available threads to compute the SpMV in such a way that different computations are assigned to each thread. This distribution depends on the number of threads, and diffe-

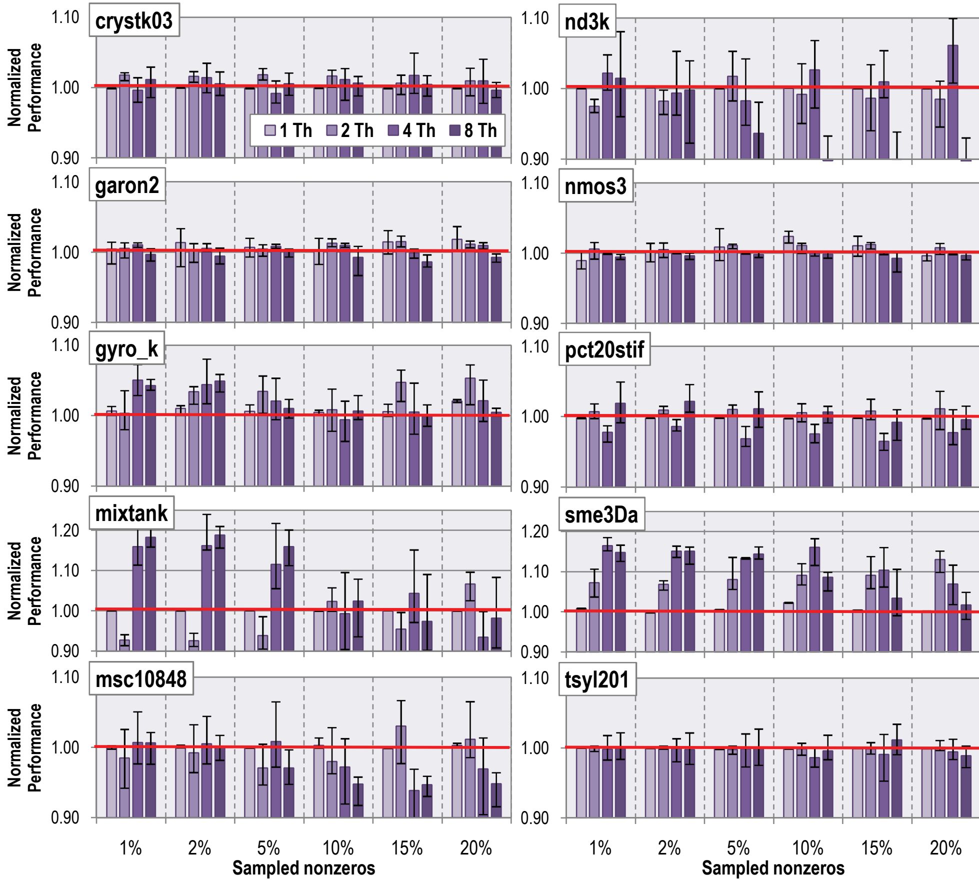



**Figure 4.5:** Normalised SpMV performance obtained by the reorderings generated using the locality optimisation technique ($w = variable$) and the information provided by the randomly sampled matrices on the Itanium2 platform.

rent memory accesses are required by each thread to perform the computations. Therefore, depending on the accesses performed by each thread, the locality optimisation technique will produce different results. Note that the performance of the parallel SpMV is determined by the slowest thread. The load balance can also influence the results for different number of threads.

Some of the previous observations for $w = 1$ coincide with the behaviour of the reorderings using windows of variable size (Figure 4.5). Note that considering only 1% of nonzeros is

again enough to generate reorderings that show similar performance compared to the original ones. However, the results show a variability higher than the results in Figure 4.4. For example, considering few sampled nonzeros, noticeable improvements are achieved by the `mixtank_new` reorderings using four and eight threads, while there is some degradation with two threads. An irregular performance is also obtained by the reorderings of matrices `nd3k`, `msc10848` and `sme3Da`. Additionally to this, there are some cases in which reorderings based on sampled information always outperform those obtained by the original technique. This is the case of matrices `gyro_k` and `sme3Da`. We find the cause of this performance variability in the variable-size windows creation process of the locality optimisation technique [12]. Note that the criterion used to decide whether two consecutive rows/columns fall into the same window depends on the locality estimation performed by the model. Therefore, different windows of locality (both in number and composition) are considered to calculate the permutation vector for each sampled matrix, which causes more variations in the performance results than using windows of fixed size.

To summarise, this study demonstrates that it is feasible to perform a data reordering to optimise the locality of the SpMV considering only a subset of the nonzeros of the sparse matrix. Tested using windows of fixed and variable sizes, it proves that a few number of nonzeros (typically 1-2%) is enough to obtain a similar performance with respect to the original reorderings. It must be noticed, however, that important variations in the performance within each subset were observed when using randomly sampled matrices with the reordering technique. These fluctuations led, in some cases, to a variation higher than 15% between the maximum and the minimum performance.

## 4.4 Locality optimisation using hardware counters

The second method developed to obtain a subset of nonzeros from a sparse matrix, in order to reduce the processing time of the reordering technique, involves the use of the hardware counters provided by the Itanium2 processor. This section introduces and evaluates a novel technique to characterise a matrix using events captured by the Event Address Registers (EARs). We have named this technique DAST (*Dual Array Sampling Technique*). Before introducing it, a couple of facts must be revisited:

– `Perfmon` performs an event-based sampling (EBS). That is, every time a given event occurs, the counter counting the number of captured EAR events increments its value

until it overflows. When this happens, a sample with information about the event that triggered the overflow is collected and stored in a buffer. The information collected comprises, among others, the latency and the address of the memory access sampled. Once the buffer gets full, an interruption is raised and the content of the buffer is available at user level. Note that `Perfmon` makes possible to monitor a particular memory address range of the data accessed by the monitored program. For more detailed information about the sampling process, see Section 3.2.

– Section 2.3.1 explained that floating-point operations on Itanium2 systems bypass the L1 cache in such a way that every access to a floating-point value always generates a L1 cache miss. An access to any data stored in a L2 cache has, at least, a 5-cycle latency for integers and 6 cycles for floating-point data. Therefore, setting the PMU to capture *"cache misses with latencies higher than 4 cycles"* (event `DATA_EAR_CACHE_LAT4`) guarantees that all accesses to the arrays $X$ and $Y$ in the SpMV are susceptible of being sampled.

A priori, by just getting the address of an access to the array $X$ in the SpMV, the position of the accessed nonzero element in the matrix cannot be determined. The reason is that the counters only provide the latency and address of the $X$ element that misses the cache. That is, only information about the column of the corresponding nonzero element of the matrix is given. Therefore, the identification of the row is not possible by using only this information.

DAST overcomes this limitation and obtains the row and the column of an access in the following manner: `Perfmon` is configured to monitor accesses to arrays $X$ and $Y$ in the SpMV. Therefore, some of the sampled events will be caused by accesses to $X$ and some of them by accesses to $Y$. Note that accesses to $Y$ are driven by the loop $i$ in the code of Algorithm 1. After a monitoring period, defined as the time between two consecutive buffer overflows –which comprises several hardware counter overflows, related to the configured sampling period–, EARs will provide a list of sampled accessed elements from $X$ and $Y$. For example, let us consider seven sampled events so that the result of the captured accesses are: $Y[0]$, $X[23]$, $Y[1]$, $Y[2]$, $X[12]$, $X[19]$ and $Y[2]$. Accesses to $X$ give information about the exact column of the corresponding nonzero element of the matrix, while accesses to $Y$ give information about the rows where the nonzero element might be placed. Note that this method does not provide simultaneously the row and column of an entry of the sparse matrix. However, the nonzero elements of the sparse matrix can be characterised according to the following properties:

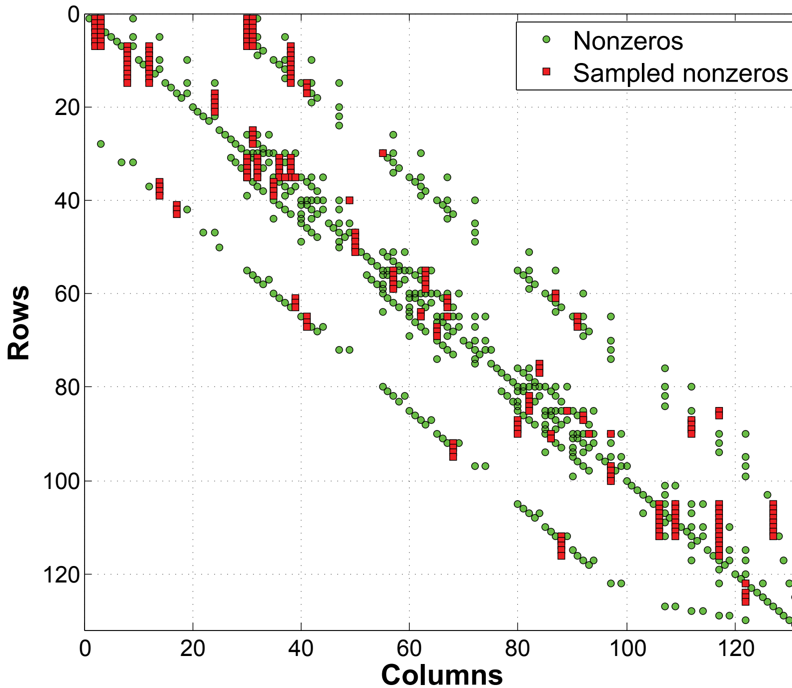



**Figure 4.6:** Example of sampled matrix using the hardware counters.

- Samples fetched from the buffer are known to have been captured in chronological order.
- Each sampled access to $Y[i]$ indicates that any nonzero element that belongs to the row $i$ of the matrix is being multiplied by one element of vector $X$.
- Each sampled access to $X[j]$ indicates that a nonzero element of the matrix belonging to the $j$ column has been accessed.
- We can state that the matrix has nonzero elements in the columns provided by the indices of the sampled $X$ elements collected between two samples $Y[i]$ and $Y[i'], i' \geq i$. These nonzero elements are located in the rows within the interval $[i, i']$. From now on, these intervals are denoted as *uncertainty intervals*. In the example above, there is an entry in column 23 in row 0, row 1 or both, since there is an access to $X[23]$ between two accesses to $Y[0]$ and $Y[1]$.

– As a particular case of the property above, if $i = i'$ then the row is determined in an univocal way. Sampled nonzeros for which the row can be univocally determined will be called *univocal sampled entries*. In the example above, columns 12 and 19 and row 2 will give us the univocal pairs $i, j = [2, 12]$ and $i, j = [2, 19]$. This is because accesses to $X[12]$ and $X[19]$ occur between two accesses to $Y[2]$.

As a way of illustration, DAST is used with the previous seven example events. It is clear that there are three nonzeros of the matrix in the positions $([0, 1], 23)$, $(2, 12)$ and $(2, 19)$. The row in the first entry cannot be exactly stated, so an uncertainty interval of possible rows $([0, 1])$ is given. The other two cases are univocally sampled entries. An example of the pattern characterisation using this methodology with a real sparse matrix is shown in Figure 4.6 (in green circles the pattern of the matrix, and in red squares the sampled nonzeros). Note that the uncertainty intervals are shown as a sequence of nonzeros along the same column in the sampled matrix.

Therefore, by just keeping the univocal pairs row-column, we can build a sampled version of the original matrix. From now on, we refer to the matrix obtained in this way as the *sampled matrix*.

### 4.4.1 Behavioural study of DAST

DAST returns a matrix created from a univocal subset of the sampled values. This section studies the nature of this sampling. The incomplete information due to the sampling process could result in a malfunction of the algorithm to improve locality, situation that is considered an error. The possible error introduced with respect to the data pattern used and its effect to allow our test algorithms to work properly is analysed here. This study is carried out in a controlled environment in order to state which parameters of the data pattern affect the difference between the original cases and the sampled ones.

#### 4.4.1.1 Methodology

To identify the influence of different characteristics of the matrix pattern in a parameterised and controlled way, a banded matrix generator was implemented. This matrix generator has the following features:

– It generates a banded matrix by giving random values with a normal distribution. The GSL (*GNU Scientific Library*) [80] was used to obtain a set of normally-distributed random values in the band.

– The modifiable parameters are the NNZ (number of nonzero entries), the number of rows and columns, the centre and sigma ($\sigma$) of the gaussian distribution, the bandwidth and the scale factor keeping the density in the band. This last parameter scales the width of the band, keeping the amount of nonzero elements defined in the "bandwidth" parameter.

– Matrices are generated in CSR (*compressed sparse row*) format.

280 matrices were generated with the following characteristics:

– $N$ (number of rows and cols): 20000

– Bandwidth: 2000

– Band scale factor: 2

– $\mu$ (centre of the gaussian): 50

– Sigma ($\sigma$) : 14 uniformly-distributed values between 0.1 and 3.

– NNZ: 20 uniformly-distributed values between 200000 and 3848462.

Each matrix name has this nomenclature:

```
bandedMatrix_N_Bandwidth_ScaleFactor_sigma^2*100_mu_NNZ.rcs
```

Note that *standard deviation* is referred as the one derived from the average number of entries per row, and not as the $\sigma$ (sigma) value of the gaussian bell. A higher $\sigma$ implies a more spread gaussian and, therefore, a more uniform amount of nonzero values spread out along the band. Hence, an increasing value of $\sigma$ in a matrix implies a decreasing standard deviation.

This test intends to find out whether there is a relation between the irregularity of the sparse matrix (expressed as the standard deviation) and the characteristics of the sampled data. To understand this relation, the generated matrix set was firstly sampled using DAST and, then, the relationship between the sampling rate and the characteristics of each matrix was analysed.

#### 4.4.1.2 Outcomes and discussion

Figure 4.7 shows three examples for $\sigma = 0.1$ (a), $\sigma = 0.8$ (b) and $\sigma = 3$ (c). The *NNZ* values for each case are 200000 (leftmost), 1131420 (centre) and 3848462 (right). On each graph, the red histogram represents the number of nonzero elements per row of the original matrix. The blue cloud of dots shows the sampled percentage per row (one dot per row).

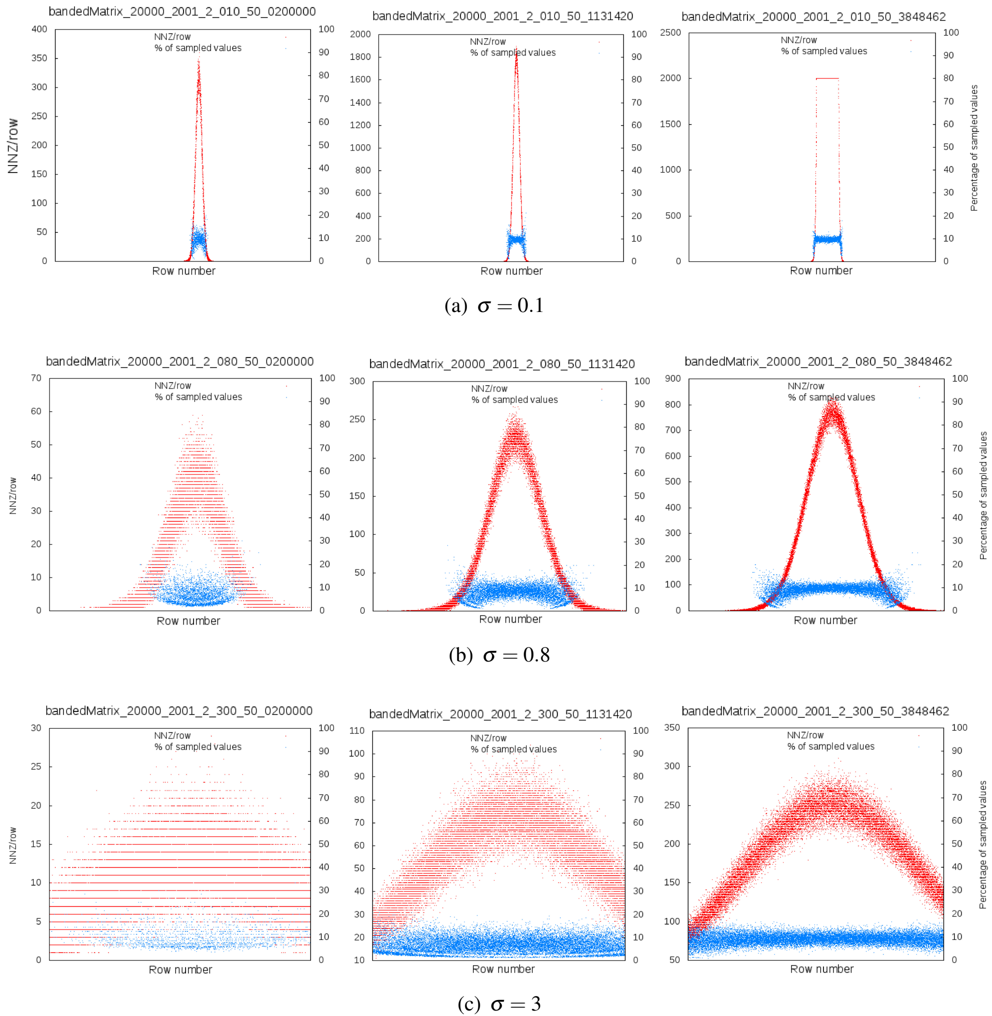
Figure 4.7(a) shows a very narrow gaussian bell ($\sigma = 0.1$). As the number of entries per row increases, the sampled ratio (in blue) converges to a more uniform 10% band (that is, more rows achieve a 10% sampling percentage) with some blurred zones in the edges of the curve. The same behaviour can be observed for a wider bell of $\sigma = 0.8$ (Figure 4.7(b)). In this case, the sampled ratio also converges to a 10% band, but it starts with a more sparse sampling and has most data sampled only on the centre of the bell, where the density is higher. On the contrary, Figure 4.7(a) always had the whole band sampled for all cases because there were enough nonzero values per row. In the third case (Figure 4.7(c)), the density is more uniform along the band, so the number of nonzero values per row is therefore lower. In this case, only the rightmost figure shows a uniform 10% sampled ratio, when there are enough nonzero values to sample.

It is worth mentioning that some cases, like the rightmost one in Figure 4.7(b), show some *"sparse cloud of samples"* on the edges of the gaussian bell. In those zones, the number of entries per row drops drastically. As a consequence, any single sample obtained in those zones gets a higher relative importance than the samples situated on the centre of the bell.

Therefore, DAST samples a matrix accurately when the number of nonzeros per row is uniform along the whole band and high enough to generate the necessary L1 misses so that enough samples are captured by the hardware counters. For other cases, the zones of the matrix that do not achieve this threshold are not so accurately sampled.

By and large, results show that the sampling is adequate except for those cases where the matrix presents several regions with a steep difference of density among them or when there is a low density per row. In these situations, DAST stresses the differences among those regions.

We found the ultimate cause for this behaviour in the way DAST works: when sampling a matrix row, only a small percentage of the samples is kept (the *univocal samples*). Take as an example the SpMV: if the considered row has not enough nonzero elements, the probability of sampling entries univocally –that is, one or more elements of $X$ between two values of the same $Y$–, decreases as the density of the row does. Therefore, those matrices with a low number of entries per row will get less samples than the rest of them.




(a) $\sigma = 0.1$

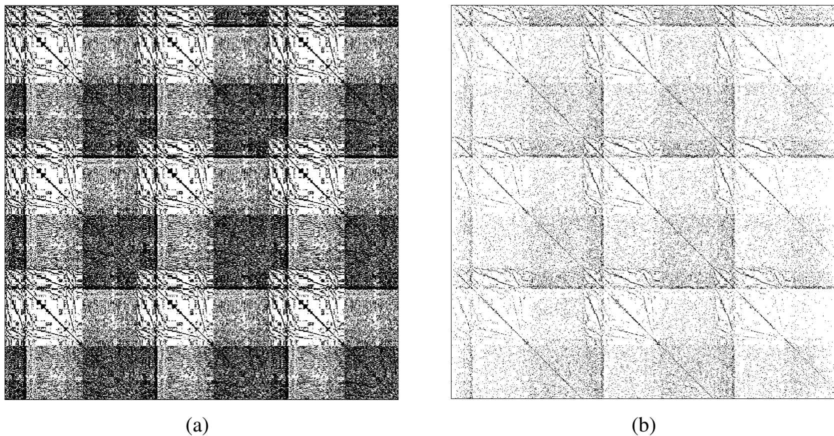
(b) $\sigma = 0.8$

(c) $\sigma = 3$

**Figure 4.7:** Entries per row (red) and sampled percentage (blue) of nine generated matrices.

| *Matrix* | # *sampled nonzeros*(*snnz*) (% *w.r.t. nnz*) | # *snnz*/*n* | #*rows without sampled nonzeros*(% *w.r.t. n*) |
|---|---|---|---|
| *crystk*03 | 128394 (7.3%) | 5.2 | 754 (3.0%) |
| *garon*2 | 17250 (4.4%) | 1.3 | 5859 (43.3%) |
| *gyro_k* | 70405 (6.9%) | 4.1 | 2342 (13.5%) |
| *mixtank_new* | 143226 (7.2%) | 4.8 | 4630 (15.5%) |
| *msc*10848 | 101347 (8.2%) | 9.3 | 85 (0.8%) |
| *nd*3*k* | 309116 (9.4%) | 34.3 | 0 (0 %) |
| *nmos*3 | 11249 (2.9%) | 0.6 | 11111 (59.8%) |
| *pct*20*stif* | 176469 (6.5%) | 3.4 | 6927 (13.2%) |
| *sme*3*Da* | 63984 (7.3%) | 5.1 | 758 (6.1%) |
| *tsyl*201 | 204825 (8.3%) | 9.9 | 29 (0.1%) |

**Table 4.2:** Characteristics of the sampled matrices generated by the hardware counters.



(a) (b)

**Figure 4.8:** `sme3Da` original (a) and sampled matrix generated by the hardware counters (b).

### 4.4.2 Results and evaluation using DAST

Table 4.2 shows the number of elements sampled after running a SpMV using DAST on the sparse matrices from the testbed of Table 4.1. Only the univocal sampled nonzeros are considered. The number (*snnz*) and the percentage of univocal sampled entries with respect to the number of nonzero elements of the original sparse matrix are displayed. This percentage ranges from 2.9% (matrix `nmos3`) to 9.4% (matrix `nd3k`). The number of univocal sampled nonzeros per row is also shown in the table. Figure 4.8 illustrates an example of matrix

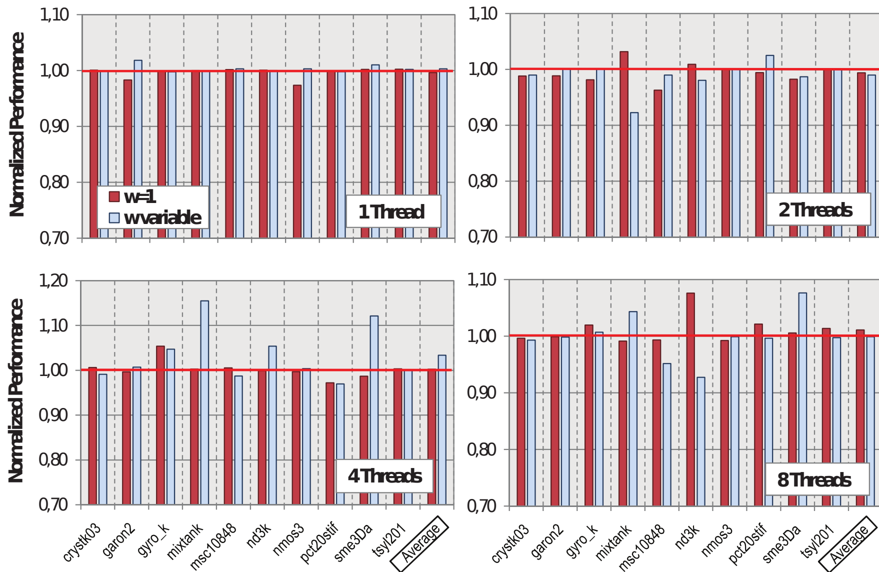



**Figure 4.9:** Normalised SpMV performance obtained by the reorderings generated using the locality optimisation technique and the information provided by the hardware-counter sampled matrices for windows of size $w = 1$ and variable size.

sampled using the hardware counters. In particular, the sampled matrix contains 7.3% of the nonzero elements of the original sme3Da matrix.

We have observed that using this sampling method, in some cases, there are rows without sampled nonzeros. Table 4.2 shows its number and percentage with respect to $n$ for each sampled matrix. We demonstrated that these percentages remain constant regardless of the SpMV iteration in which the sampling is performed. However, the values obtained show a strong dependence with the matrix pattern. The lower the number of nonzeros per row in the original matrix (see $nnz/n$ in Table 4.1), the higher the number of rows without sampled entries. For example, matrix nmos3 has almost 60% of its rows without sampled entries, whereas for nd3k every row has, at least, one sampled element.

Figure 4.9 shows a comparison between the performance of the SpMV obtained by the original locality optimisation technique and the performance obtained using the information provided by the hardware counters. As in Section 4.3, the information provided by the sampled matrices has been used to generate a permutation vector which is applied to the original

matrix. In this way, the original matrix is reordered considering only the information provided by a subset of its nonzeros.

The SpMV performance of the sampled information is normalised to the performance of the original technique. Reorderings using windows of fixed size with $w = 1$ and variable size are analysed. In most of the cases, reorderings guided by the information provided by the sampled matrices achieve a very similar performance to those obtained by the original technique. A few exceptions can be noticed. For example, considering windows of fixed size, the original `pct20stif` reordering outperforms the sampled one when using four threads. The differences in the performance for these cases are, at most, about 3%. On the contrary, there are some sampling-based reorderings that achieve better performance compared to the original ones. This is the case of the matrix `nd3k` using eight threads, showing an improvement of 8%. A similar behaviour is observed considering windows of variable size. Note that the magnitude of the performance variations for some matrices is slightly higher in comparison with the fixed-sized case. For example, an improvement of about 15% is achieved by the `mixtank_new` sampling-based reordering using four threads.

It is noticeable that, unlike the use of random sampling (Section 4.3), the performance obtained with this technique shows smaller fluctuations (always lower than 2%). This is due to the kind of EBS method used by `Perfmon`. In EBS, a sampling period is expressed as a number of occurrences of an event. According to [67], using a fixed sampling period may easily lead to biased results. This is the reason why `Perfmon` can use a randomisation of the sampling periods. After each buffer overflow, `Perfmon` randomises the moment in which monitoring is resumed. In this way, sampling based on hardware counters performs a uniform EBS but does not use a fixed sampling period.

According to the previous results, it can be concluded that the sampled information provided by the hardware counters is enough for the locality optimisation technique to generate quality reorderings. This upholds the observations from Section 4.3.

### 4.4.3 Comparison with the original non-reordered matrices

Up to this point, the SpMV performance of matrices reordered from the sampled information provided by DAST has only been compared with the reorderings generated by the original technique. This section compares it to the performance of the SpMV using the original matrices (without reordering), in order to check whether sampling-based reorderings obtain a better SpMV performance. Results of this study are shown in Table 4.3.

| *Matrix* | 1 *Thread* | | | 2 *Threads* | | |
|---|---|---|---|---|---|---|
| | Orig. | w=1 | w=var. | Orig. | w=1 | w=var. |
| *crystk*03 | 0.36 | **0.38** | **0.37** | 0.47 | **0.50** | **0.49** |
| *garon*2 | **0.53** | 0.52 | **0.53** | 1.18 | **1.21** | **1.21** |
| *gyro_k* | 0.33 | **0.34** | **0.34** | 0.70 | **0.79** | **0.76** |
| *mixtank_new* | 0.33 | **0.35** | **0.35** | **0.52** | 0.48 | 0.48 |
| *msc*10848 | 0.33 | **0.35** | **0.35** | 0.43 | **0.59** | **0.60** |
| *nd*3*k* | 0.33 | **0.35** | **0.34** | **0.50** | 0.47 | 0.46 |
| *nmos*3 | 0.48 | **0.49** | **0.50** | 1.08 | **1.10** | **1.09** |
| *pct*20*stif* | 0.35 | **0.37** | **0.37** | 0.47 | 0.47 | **0.48** |
| *sme*3*Da* | 0.35 | **0.37** | **0.37** | 0.86 | **0.91** | 0.83 |
| *tsyl*201 | 0.31 | **0.32** | **0.32** | 0.43 | **0.44** | **0.44** |
| *Average* | *0.37* | *0.38* | *0.38* | *0.66* | *0.70* | *0.68* |
| *Matrix* | 4 *Threads* | | | 8 *Threads* | | |
| | Orig. | w=1 | w=var. | Orig. | w=1 | w=var. |
| *crystk*03 | 1.64 | 1.64 | **1.70** | 5.22 | **5.32** | **5.31** |
| *garon*2 | 2.28 | **2.38** | **2.39** | 3.73 | **4.66** | **4.66** |
| *gyro_k* | 2.40 | **2.47** | **2.57** | 5.12 | **5.31** | **5.36** |
| *mixtank_new* | 1.03 | **1.38** | **1.42** | 2.29 | **4.79** | **4.16** |
| *msc*10848 | 2.00 | **2.54** | **2.44** | 5.00 | **6.08** | **5.68** |
| *nd*3*k* | 0.77 | **0.89** | **0.93** | 2.42 | **3.22** | **2.71** |
| *nmos*3 | 2.15 | **2.18** | **2.18** | 4.16 | **4.26** | **4.27** |
| *pct*20*stif* | 0.92 | **0.99** | **0.99** | 3.73 | **3.76** | **3.74** |
| *sme*3*Da* | 2.61 | **2.68** | **2.67** | 4.79 | **5.64** | **5.25** |
| *tsyl*201 | 1.05 | **1.06** | 1.05 | 4.51 | **4.54** | 4.51 |
| *Average* | *1.68* | *1.82* | *1.83* | *4.09* | *4.76* | *4.57* |

**Table 4.3:** SpMV performance comparison between the original matrices and the reorderings obtained by the locality optimisation technique (in GFLOPS). Reorderings were performed using the information provided by the hardware counters.

At first sight, the results point out that matrices obtained after applying the data reordering technique outperform the original ones. There are only a few cases (using one and two threads) that do not benefit from this reordering. SpMV performance improves up to 53%, which is the case of matrix `mixtank_new` running with eight threads. Note that, in this example, the permutation vector can use only 7.2% of the nonzeros of the matrix (see Table 4.2). Note also that, as the number of threads increases, the performance improvement caused by the locality optimisation gets more important. For example, considering windows of fixed

size ($w = 1$), the performance increases on average from 2.6% in the sequential case to 14.1% with eight threads. Finally, a slightly better behaviour is observed for reorderings that use windows of fixed size instead of windows of variable size. This observation agrees with the conclusions in [12].

### 4.4.4 Overhead improvement

One of the main drawbacks of the data reordering techniques is the reordering cost. It can only be amortised when the sparse operation is repeatedly performed as, for instance, in iterative methods, which usually require hundreds or even thousands of sparse matrix-vector multiplications [79]. As shown next, a consequence of using sampled information to calculate the permutation vector is an important reduction in the overhead introduced by the reordering technique.

The reordering technique used in this study has two stages: the graph calculation and the generation of the permutation vector using the TSP heuristic. In both cases the reduction in the number of nodes of the graph decreases noticeably their overhead. If the reduction is performed in the number of edges, the overhead also decreases, but to a lesser degree. Using sampled matrices, the number of edges are reduced and the edge weights are changed in the graph. Note that there will be an edge between two nodes when there is, at least, one

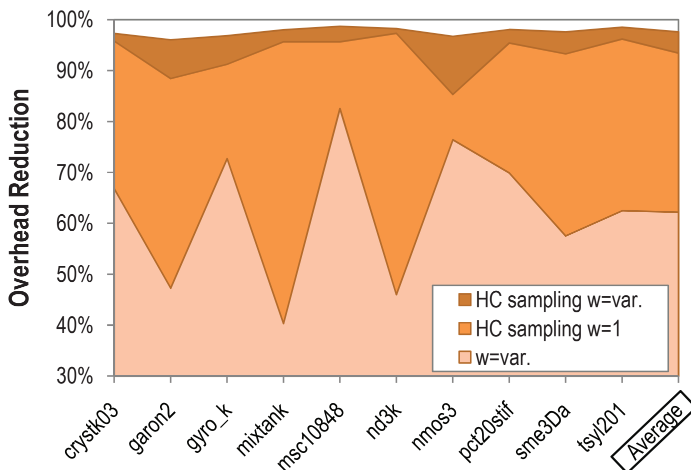



**Figure 4.10:** Overhead reduction of the locality optimisation technique using as a reference the time required to perform the reordering using windows of fixed size $w = 1$ and the original (non-sampled) matrices.

entry match (see Section 4.2). Given that sampled matrices have fewer nonzeros than the original ones, edges between nodes with few entry matches in the original graph are bound to disappear in the graph generated using the sampled matrices. Note that these edge weights, according to the distance function, represent the worst cases regarding locality. Therefore, sampling reduces the cost of building the distance graph and its size. We must highlight that this distance graph is used as input of the TSP heuristic (Chained Lin-Kernighan algorithm). This heuristic is limited to a fixed number of iterations in such a way that most of the time is devoted to the graph calculation. TSP heuristic time dominates the overhead only when considering small matrices.

The analysis of the overhead is displayed in Figure 4.10. The plot shows the reduction in the reordering cost using as reference the time required by the technique when using windows of fixed size $w = 1$ and the original non-sampled matrices.

When using windows of variable size and non-sampled matrices, the locality optimisation technique reduces on average 62% the reference overhead. But this reduction is even more noticeable when sampled matrices are considered. In this case, the overhead reduction reaches 93% and 98% on average, depending on whether fixed or variable size windows are used, respectively.

In a more precise way, the overhead expressed in terms of the number of SpMV operations and considering sampled matrices is on average 620 with $w = 1$, and 261 with $w = variable$. Note that the reference here is the computational time required to perform the SpMV operation on the original matrices.

According to these results, we conclude that similar performance improvements are achieved considering only sampled information with respect to the original locality optimisation technique with an important reduction in the computational time required to perform the reordering.

## 4.5 Locality optimisation using latency information

Additionally to the addresses that cause a particular event, EARs provide the latency of every sampled access to memory. It should be remembered that, using our sampling method, when a sampled nonzero of the matrix is multiplied by the corresponding element of the vector $X$, the access latency of this element becomes available. This latency information can be used to improve the behaviour of the data reordering technique. With this objective in mind, several new criteria to create the windows of locality are introduced in this section.

In [12], windows of variable size are built according to the locality model. That is, considering only the distances among the rows/columns of the original sparse matrix. Now, a new approach that uses the maximum sampled latency per row instead is introduced. We assume that if there are accesses with high latency to the vector $X$, there is a low data reuse of these elements accessed by the nonzeros of a particular row and the ones accessed by the previous rows. Taking this into account, the windows creation process can be divided into two stages:

1. A histogram is created from the input matrix, that is, the sampled one. This histogram only contains the maximum latency per row. Therefore, there are $n-1$ latency values in the histogram. Note that all the information provided by the hardware counters is obtained in the second iteration of the SpMV, to avoid high latencies caused by the compulsory misses.

2. The decision of whether two consecutive rows $i$ and $j$ will be included within the same window or not is ruled by the following condition:

$$maximum_sampled_latency(j) \leq threshold \tag{4.5}$$

   Otherwise, a new window is created. The behaviour of two different thresholds has been evaluated: average and 7-cycle latency. The average latency depends on the sampled accesses and it is calculated as:

$$\frac{\sum_{i=0}^{snnz} latency(\beta_i)}{snnz} \tag{4.6}$$

   where $\beta_i$ is a sampled access. Note that latencies higher than 7 cycles correspond, at least, with accesses to the L3 cache on the Itanium2 processor.

Once the windows are defined, the locality optimization technique generates a permutation vector that will be subsequently applied to the original matrix. Note that this permutation vector is calculated by our reordering technique using only the information provided by the sampled matrices and the windows obtained considering the latency information.

On the other hand, we have detected that some rows of the matrices have no nonzero values sampled by the hardware counters (see Table 4.2). Therefore, there is no latency information for some rows of the sampled matrices. This situation implies a problem in the windows creation process: how can we handle rows without sampled entries?. Several tests have been performed to figure this out:

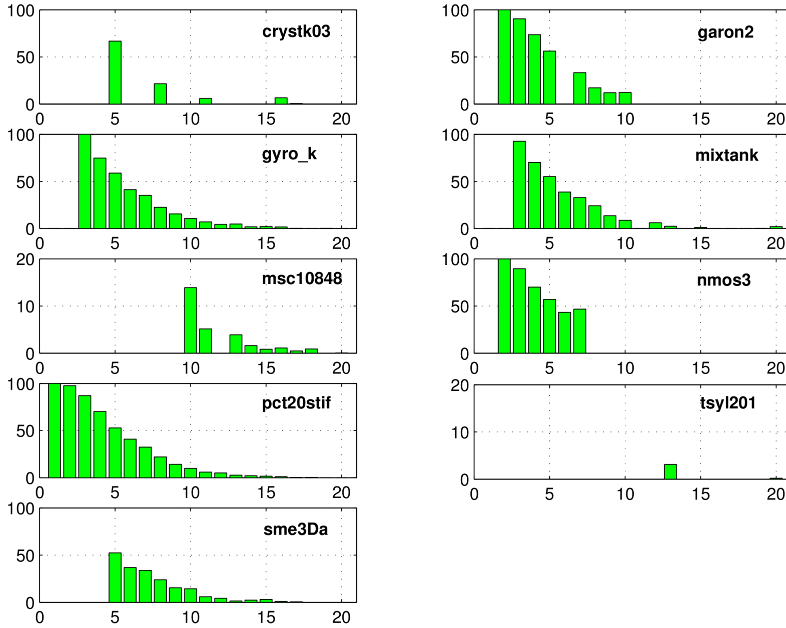



**Figure 4.11:** Figures display, for a particular row size interval ($X$ axis), the percentage of these rows with latency=0 (without sampled nonzeros) in the latency histogram ($Y$ axis). Interval $a$ corresponds to rows of the matrix with $[5a, 5(a+1))$ nonzeros.

– Firstly, an analysis of the latency histograms was performed. Those matrices that did not get any sampled access at any of their rows were checked to find out their number of nonzero elements per row. A summary of the results is shown in Figure 4.11. The figures display, for a particular row size interval ($X$ axis), the percentage of these rows with latency zero in the histogram ($Y$ axis). In particular, interval $a$ corresponds to row sizes in the interval $[5a, 5(a+1))$. For example, interval number 5 corresponds to rows of [25,30) nonzero elements. Interval number 21 shows the percentage for rows with more than 150 nonzero elements. Results show the same trend for all the considered cases. That is, the greater the number of nonzero elements per row, the smaller the percentage of rows with no sampled accesses.

– Secondly, a comparison between the latency histograms of several SpMV iterations was performed. For those rows with no sampled accesses in the 2nd iteration, the latencies in subsequent iterations were checked. Table 4.4 shows, as an example, the comparison

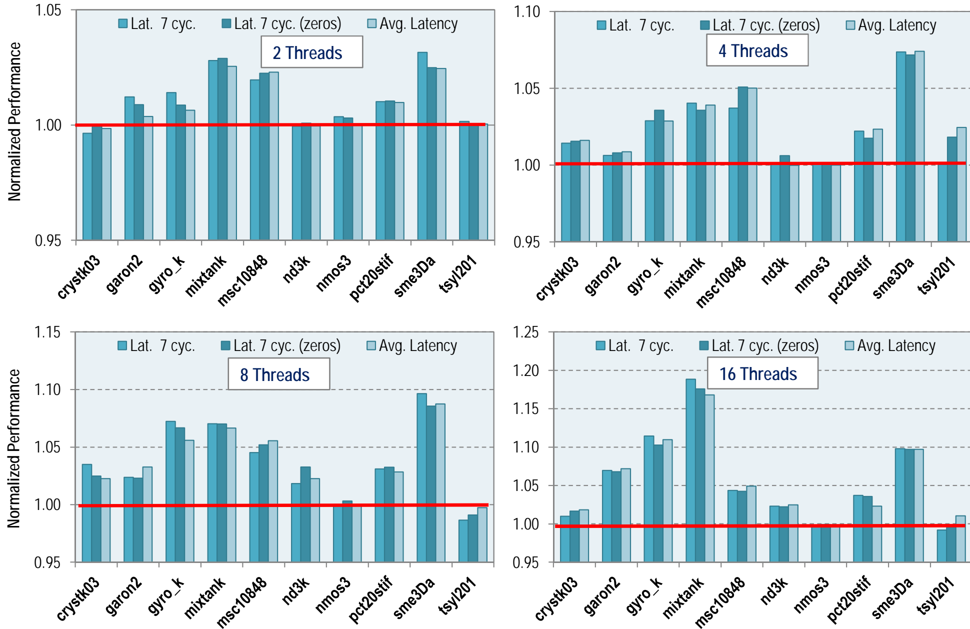



**Figure 4.12:** Normalised performance of the matrices reordered using latency information with respect to the original reordering technique.

among iterations 2, 4 and 6. This table shows the maximum latencies in iterations 4 and 6 of those rows with no sampled entries in the second iteration. Rows are grouped by its latency (L2 cache, L3 cache and memory accesses respectively). Note that most of the rows with no sampled entries in iteration 2 have low latencies in iterations 4 and 6 (typically, lower than 15 cycles). Just in a few cases (matrices `gyro_k`, `mixtank_new` and `pct20stif`) there are some rows with high latencies.

According to these analysis we conclude that rows with no sampled entries in the latency histogram (i.e., latency=0) are bound to present small size and low latency. Therefore, a good approximation will be to include those rows with no sampled entries together with its previous row in the same window of locality.

The next step consisted in evaluating the SpMV performance of the matrices reordered using the latency information. Two thresholds were used: average latency and 7-cycle latency. In both cases, a new window is created for each row with no sampled accesses. Additionally

| $Matrix$ | $Iter.$ | $Latency\ interval\ (cycles)$ | | |
|---|---|---|---|---|
| | | $[5,7]$<br>L2 cache | $(7,21]$<br>L3 cache | $> 21$<br>Memory |
| $crystk03$ | | | | |
| | $4^{th}$ | 662 | 11 | 0 |
| | $6^{th}$ | 652 | 17 | 2 |
| $garon2$ | | | | |
| | $4^{th}$ | 2583 | 78 | 0 |
| | $6^{th}$ | 2466 | 73 | 0 |
| $gyro_k$ | | | | |
| | $4^{th}$ | 1557 | 9 | 3 |
| | $6^{th}$ | 1510 | 11 | 3 |
| $mixtank_new$ | | | | |
| | $4^{th}$ | 2577 | 121 | 3 |
| | $6^{th}$ | 2607 | 172 | 4 |
| $msc10848$ | | | | |
| | $4^{th}$ | 72 | 2 | 0 |
| | $6^{th}$ | 77 | 5 | 0 |
| $nd3k$ | | | | |
| | $4^{th}$ | 0 | 0 | 0 |
| | $6^{th}$ | 0 | 0 | 0 |
| $nmos3$ | | | | |
| | $4^{th}$ | 3974 | 154 | 0 |
| | $6^{th}$ | 4059 | 167 | 0 |
| $pct20stif$ | | | | |
| | $4^{th}$ | 4452 | 36 | 16 |
| | $6^{th}$ | 4570 | 34 | 18 |
| $sme3Da$ | | | | |
| | $4^{th}$ | 556 | 88 | 0 |
| | $6^{th}$ | 581 | 56 | 0 |
| $tsyl201$ | | | | |
| | $4^{th}$ | 27 | 0 | 0 |
| | $6^{th}$ | 29 | 0 | 0 |

**Table 4.4:** Comparison between the latency histograms of SpMV iterations 2, 4 and 6.

to this, a case was included for the 7-cycle threshold (labelled as “Lat.7 cyc (zeros)” in the figures) where rows with no sampled entries are included in the previous window, according to the conclusion above.

Figure 4.12 shows the SpMV performance obtained by the latency-based reorderings using different number of threads. These measurements are normalized with respect to the performance obtained by the original technique when considering windows of variable size. According to the results several conclusions can be made. Firstly, the new approach clearly outperforms the original technique. Only in a few cases the latency-based reorderings degrade the performance of the original ones. These degradations are always lower than 1%. On the contrary, speedups up to 1.18× were reached (matrix `mixtank` with 16 threads). Secondly, the impact on the SpMV performance caused by the locality improvement is more noticeable with a higher number of running threads. This is caused by the weight of the cache and memory accesses in the total execution time of the parallel SpMV kernel, which increases as the number of threads grows. In this way, for example, the highest improvements with respect to the original reorderings were observed with 16 threads. And finally, as we expected, a slightly better behaviour is observed when the *zeros* approach is not applied. Recall that with the zeros approach those rows with no sampled entries are included together with its previous row in the same window of locality. We assume that rows with no sampled nonzeros in the latency histogram are likely to present small number of nonzeros and low latency. However, in a few cases, the differences in the SpMV performance is important. This is the case of matrices `gyro_k` and `pct20stif`, whose sampled matrices have 13.5% and 13.2% of their rows without sampled entries respectively (see Table 4.2). Here windows of locality are too big in the sense that they consist of consecutive rows that exhibit low locality among them.

Therefore, according to these results, we conclude that using the latency information in the windows creation process improves noticeably the behaviour of the reordering technique in comparison with the original version.

Another important issue was studied: the overhead introduced by the reordering technique. This overhead is related to the number of nodes in the distance graph (i.e., the number of windows of locality) that is used as the input for the reordering technique. Table 4.5 shows the number of windows of locality for each matrix when different criteria are considered. Note the important reductions obtained when the latency information is used, especially when rows without sampled entries are not considered as new windows. As Figure 4.13 shows, this policy has a direct influence in the computational time required to perform the reordering. The reordering time is normalised with respect to reorderings performed using the METIS library

| *Matrix* | *N* | *Original Technique* | *Lat. 7 cycles* | *Avg. Latency* | *Lat. 7 cycles (zeros)* |
|---|---|---|---|---|---|
| *crystk03* | 24696 | 12598 | 1598 | 1508 | 760 |
| *garon2* | 13535 | 5266 | 6158 | 6158 | 301 |
| *gyro_k* | 17361 | 7144 | 2837 | 2837 | 497 |
| *mixtank_new* | 29957 | 11523 | 7211 | 7211 | 2583 |
| *msc10848* | 10848 | 4463 | 793 | 793 | 708 |
| *nd3k* | 9000 | 4765 | 1122 | 838 | 1122 |
| *nmos3* | 18588 | 6221 | 11380 | 11380 | 271 |
| *pct20stif* | 52329 | 22868 | 7856 | 7856 | 929 |
| *sme3Da* | 12504 | 4429 | 2988 | 2988 | 2230 |
| *tsyl201* | 20685 | 9324 | 1308 | 852 | 1279 |

**Table 4.5:** Number of windows of locality using different criteria in the windows creation process.

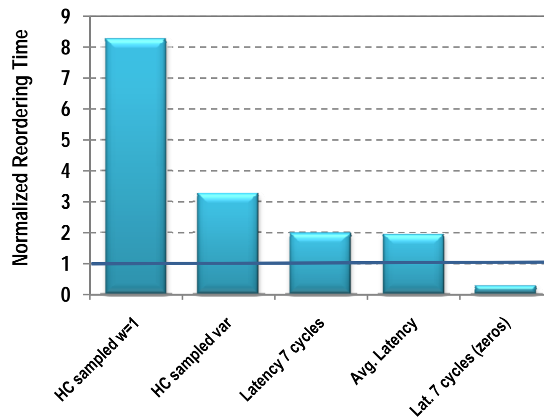



**Figure 4.13:** Average overhead of the reordering technique: comparison with METIS library.

[81], which is considered a standard. The first two bars correspond to the original reordering technique using windows of $w = 1$ and variable size respectively. The others represent the reordering technique using latency information and different thresholds. Important reductions are observed in comparison with the original technique. It is especially noticeable the last case, reducing on average a 60% the time required by METIS. The figure summarises the results for all the matrices in our testbed.

Nonetheless, every architecture has its own peculiarities and, although this new approach using latencies outperforms the original reordering technique, the latter has not been yet tested to prove that actually improves the performance of the original SpMV. Figure 4.14 shows the

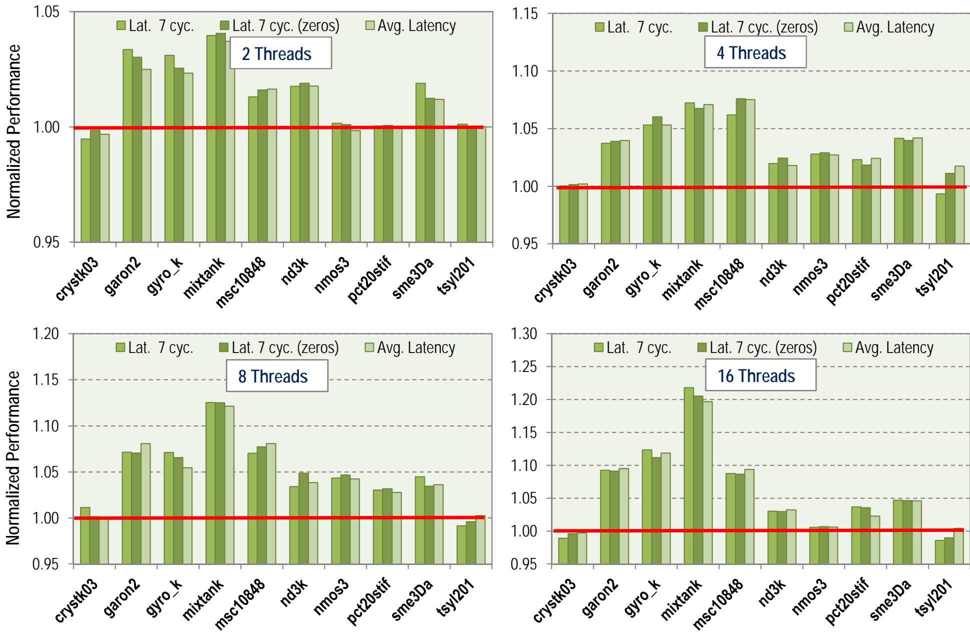



**Figure 4.14:** Normalised performance of the matrices reordered using latency information with respect to the original matrices.

normalised performance obtained by the latency-based reorderings with respect to the original matrices (non-reordered). A first approach to the results points out that matrices obtained after applying the data reordering technique outperform the original ones. There are only a few cases that do not take profit from this reordering (for example, matrix `crystk03`). It is worth mentioning that speedups higher than 1.2× were reached. On the other hand, as the number of threads increases, improvements caused by the locality optimization become more important. For example, considering a 7-cycles threshold, the SpMV performance increases on average from 2% in the 2 threads case to 7.3% using 16 threads.

To summarise, the reordering method applied together with the latency information provided by the hardware counters presents a good behaviour. Additionally, an important reduction in the computational time required by the reordering technique has been achieved.

## 4.6 Conclusions

Some of the methods used more often to optimise the locality of irregular codes in parallel architectures are run-time techniques that modify the allocation of data structures in memory. One of the major drawbacks of these techniques is the preprocessing cost of the reordering stage, which must be low enough to be efficient when used on real problems. This chapter has considered three approaches and contributed two of them to reduce this cost based on incomplete access-related input information.

The first approach has demonstrated that a randomly sampled matrix with about a 1-2% of the nonzeros can be used to perform the reordering of data with a maximum 5% loss of accuracy in the final results. The drawbacks observed are the fluctuations in performance due to the nature of the sampled information and the dependency of the performance with the number of threads. In the second approach a technique, called *DAST*, has been introduced to obtain a sampled matrix using the memory addresses of the samples given by the hardware counters on Itanium2 Montvale. The fluctuations in this case are noticeably lower, and our tests show that a percentage of samples from 2.9% to 9.4% are enough to obtain a difference, at worst, of 3% with respect to the original performance. In some cases, even an improvement in the performance is obtained. The main benefit shown by our results is a time reduction that ranges between 93% and 98% in the reordering stage. The third approach combines DAST and the memory access latency information provided by the hardware counters. Using this information, windows of locality are defined to assist the reordering stage. Performance improvements up to 20% are obtained using this technique. This approach shows an encouraging 60% time reduction compared with the standard METIS library.

Thus, this confirms that the hardware counters are an adequate vehicle to obtain sampled information at a low cost as well as contribute to increase the performance of locality techniques for irregular codes by reducing dramatically their computational time (goal G3-REORDTECHIMPRV).

# 5

# Page Migration

## 5.1 Introduction

On a NUMA environment, the adequate placement of data is essential to improve the performance of a code. A multithreaded application may vary its number of threads or have them migrated to different cores during its lifecycle, which will change its page affinity requirements (i.e., the cell to which a page is bound).

Thread migration and preemption are common events in non-dedicated environments. The available resources often must be shared among several jobs from different users or processes from the system itself, such as demons. In those situations in which a processor must run several threads in the same processor, the scheduler of the operating system may intervene preempting some threads or migrating them to free processors, in an attempt to distribute the available resources among jobs. In addition, depending on the type of parallel program, the computation of a preempted thread can be taken up by another thread. An undesirable consequence of these data movements in a NUMA system is that a thread may be placed far (in terms of access cost) from its dataset.

Not until recently did Linux gain NUMA awareness. Its first-touch page migration policy can sometimes be beneficial when the scheduler migrates a thread to a remote cell. In this case, if its dataset has not previously been touched by any other thread from a different cell, the recently migrated thread will have it allocated in its own cell. Unfortunately, this is not a common situation in many OpenMP programs, in which a master thread usually touches the whole data prior to involving the rest of threads in the parallel computation. When this happens, the migrated thread will need to access its dataset from a remote cell, with the resulting increase in the latency.

Even in those cases in which the first-touch policy allocates each thread and its dataset close to each other, any page can happen to be accessed by several threads scheduled in different cells. This is particularly true for irregular codes. In these situations, a dynamic migration policy can mitigate the problem of efficiently accessing data, as we demonstrate in this chapter.

The thread and memory allocation study developed in the FINISTERRAE supercomputer, described in Chapter 2, found a non-deterministic behaviour in the kernel's thread scheduler. Indeed, when running a multithreaded program in a rx7640 node (see Figure 2.2), some of the threads stay in the same cell and the rest are migrated to the other one, although this situation can vary at any moment during the program execution due to the appearance of other jobs in the system. By and large, the scheduler strives to allocate every thread as far as possible from each other. This is actually a fair policy, as long as each dataset is kept with its thread.

Additionally, Chapter 2 concluded describing an example scenario in which a static, first-touch allocation policy can be detrimental to the performance of a parallel program. This highlighted the importance of having each dataset placed as close as possible to its thread, even when this thread is migrated. Concerning this issue, one of the contributions of this dissertation, covered in this chapter, is the development of optimisation strategies for page migration. This task has been accomplished through the definition of a dynamic page migration scheme based on a run-time sampling of the memory references made by a monitored application. This scheme consists of three stages:

1. Development of a software infrastructure based on the information provided by the hardware counters to characterise, in terms of page accesses, the locality and affinity of a monitored application (goal G4-PAGEMIGINFR).

2. Setup of page migration policies for FINISTERRAE based on the development of new algorithms, using the information provided by our software infrastructure (goal G5-PAGEMIGALG).

3. Evaluation of effective page migration implementations.

The first stage is presented in Section 5.2. A key advantage in this development has been the availability of the information provided by EARs –such as the access latencies and exact memory addresses where events occur– in order to obtain an accurate-enough model of the memory map of a monitored application. This information sets a new pace for developing

different migration strategies in the second stage of our scheme, developed in Section 5.4. This section also evaluates the performance of each page migration algorithm proposed.

## 5.2 Development of a page migration infrastructure

### 5.2.1 Problem statement

When thinking about the prerequisites to develop a tool that provides information about page affinity and locality of a multithreaded application, as well as to take decisions to improve its performance, the Stéphane Eranian's command tool `pfmon` [82] was a reference. However, there were some drawbacks which refrained us from adopting it:

– It is solely a monitoring tool, not being able to take any decision in runtime.

– Even though it can give a histogram of cache misses as well as the accessed addresses and latencies, extracting page information from `pfmon`'s output would imply a post-process of a file's content, without having a chance to modify the behaviour of the monitored application based on that information.

– No migration algorithm can be attached to it.

– Collocation of the pfmon process cannot easily be controlled.

– It is a large and complex application, which may tamper with the performance of the monitored application.

Despite its drawbacks, `pfmon` cast some light on the requisites that our page migration infrastructure must meet:

– Attach to a multithreaded application given as a parameter. Most of the page migration techniques found in the literature require that the monitored program is linked to a given library or have its code modified somehow. The aim of our proposal is that it requires no modification of the monitored program whatsoever.

– Run in user-level, so that any user without privileges can execute it.

– Return information, at least, about the accessed pages, cache lines, latencies and CPU attached, so that there is enough information available to try different migration strategies.

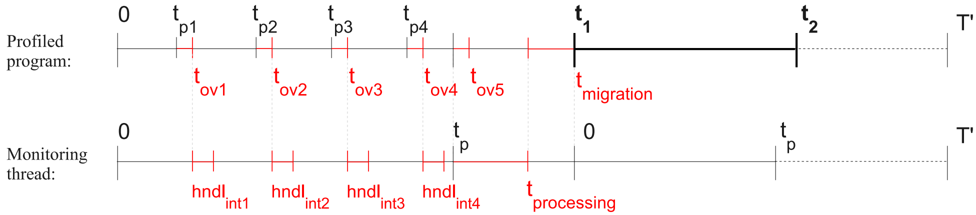



**Figure 5.1:** Time intervals in a profiling process.

– Detect and crawl over the dynamically spawned threads of the monitored application.

– Listen to every thread and show information about each of them separately.

– Monitor during the whole life cycle of a thread, or during some periods only.

– Provide flexibility to associate different migration strategies.

– Perform page migrations *timely*. That is, shortly after the need to migrate is detected.

So the combo `libpfm/Perfmon` [67] was chosen to write a program that complies with these features.

Our software infrastructure for page migration must meet two requirements: provide profiling capabilities, to inspect the data accesses in memory performed by a program in runtime, and effectively modify the allocation of such data so that the whole program locality improves. Therefore, the infrastructure must comprise two parts: a *monitoring stage* and an *evaluation stage*. Prior to developing the infrastructure, some timing constraints must be discussed.

Consider the time scenario depicted in Figure 5.1. A program is *profiled* by a *monitoring thread*. This thread uses `Perfmon` to inspect the program in a *non self-monitoring* way using a sampling buffer, procedure that was explained in Section 3.2. Each *monitoring/evaluation cycle* lasts for a time $t_i, i = 1, 2, ...n$. During the *monitoring period* of every cycle ($t_p$), the profiled program runs normally until a buffer overflow occurs at a time $t_{pj}, j = 1, 2, ...m$. For each overflow, an interruption is raised and the profiled process is stopped, with the resulting overhead ($t_{ovj}$). The interruption is then handled and the buffer is read and processed by the *monitoring thread* ($hndl_{intj}$). Then, monitoring is resumed and the whole process is repeated until the buffer overflows again at time $t_{p(j+1)}$. This process lasts until the *monitoring period* expires. In that moment, the *evaluation stage* begins. All the collected data is processed and

a migration algorithm is applied ($t_{processing}$). Next, a number of pages are migrated if needed ($t_{migration}$). Hence, each *monitoring/evaluation cycle* $t_i$ verifies:

$$t_i = \sum_j t_{pj} + \sum_j t_{ov\,j} + t_{processing\ i} + t_{migration\ i} \tag{5.1}$$

Note that equation 5.1 does not include the period $hndl_{int\,j}$, which is computed by the monitoring thread and will be masked by the monitored program execution. Only the interruption overheads and the migration time are susceptible of slowing down the monitored program.

Despite the implicit overheads related to interruption handling, processing and migrating time, and cache and TLB flushes, an adequate page allocation on each cycle $i$ is expected to amortize these costs by the locality improvement achieved. Therefore, if the original duration of a program is $T$, and its execution time using a dynamic page migration is

$$T' = \sum_i t_i \tag{5.2}$$

then the migration infrastructure will meet its goal if

$$T' < T \tag{5.3}$$

This condition has been considered in the rest of this chapter when the delays and overheads introduced by the migration infrastructure are estimated. The remainder of this section describes the page migration software infrastructure developed. In particular, the system architecture is presented in Section 5.2.2 and its functional design is explained in Section 5.2.3.

### 5.2.2 Architectural design

A controlling thread using `Perfmon` can simultaneously monitor several threads. To do so, it waits on multiple contexts[1] at the same time, as shown in Figure 5.2. A context can only be attached to one thread at a time. Therefore, there must be as many contexts as there are monitored threads.

Our page migration infrastructure takes into account this feature and comprises the modules depicted in Figure 5.3:

- A ***monitoring thread*** is allocated in a *monitoring core*, used exclusively to manage the sampling process, information retrieval and page migration decisions. This thread waits on as many contexts as monitored threads there are at a given time in a dynamic way.

[1] `Perfmon` contexts were explained in Chapter 3.

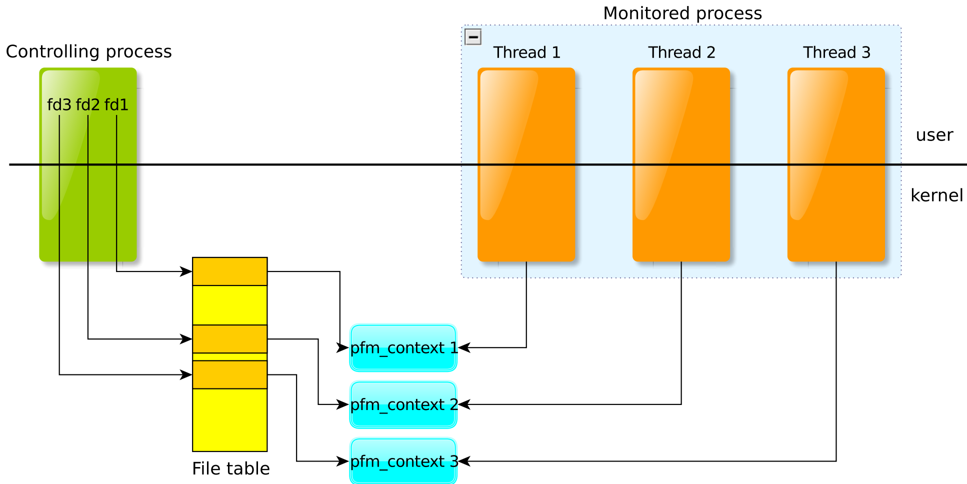



**Figure 5.2:** Waiting on multiple contexts.

- A ***multithreaded application*** where each *application thread* runs on a different core in our target architecture. A `Perfmon` context is created and associated to each thread. It permits retrieving the information provided by the PMU of each core.

- A ***sampling buffer***[2] is attached to each context, so that samples obtained by each core's PMU are stored in it.

- The sampling data obtained from each sampling buffer are processed by the monitoring thread to update a ***sample page map***. This map consists of sampled page addresses accessed per monitored thread as well as the sum of latencies of each single sampled access to a given page. Each entry in the table owns, in turn, another table with additional information such as the precise address of the sampled cache lines accessed per page, its latencies and other side information like timestamps.

[2] Sampling with `Perfmon` was explained in Chapter 3.

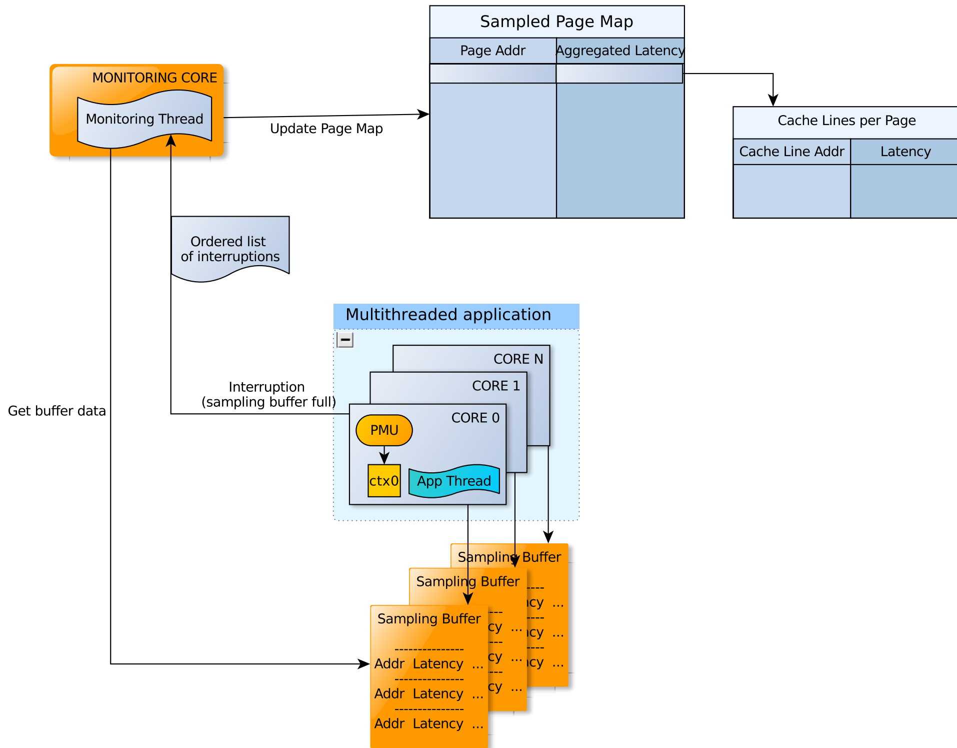



**Figure 5.3:** Sampling section from our page migration infrastructure.

### 5.2.3 Functional design

The algorithm of the *monitoring stage*, executed by the monitoring thread, is presented in the pseudocode of Algorithm 3. It comprises a main function (lines 1 to 16) and an interruption handler (lines 18 to 34). The task of the main function is to configure the monitoring process and launch the monitored process. The task of the interrupt handler consists of attending the interruptions raised by the threads of the monitored process. Both parts of the algorithm are explained next.

A flow diagram of the main function is shown in Figure 5.4. It comprises the following steps:

– The program receives the name of the application to monitor.

**Algorithm 3:** Run-time sampling infrastructure pseudocode

```
 1: main()
 2: {
 3: install_overflowHandler()
 4: initialize_pfm_library()
 5: create_child(monitored_program)
 6: for (;;) do
 7:     while (tid = new_thread_detected()) do
 8:         ctx = create_pfm_context(tid)
 9:         addContext2list(ctx)
10:         perfmonctl(PFM_LOAD_CONTEXT, tid)
11:         ptrace(tid)
12:         task_start(tid)
13:         activeThreads++
14:     end while
15: end for
16: }
17:
18: overflowHandler(tid)
19: {
20: msg = read(tid)
21: switch(msg.type){
22:   case PFM_MSG_OVFL :  /*sampling buffer is full*/
23:     process_smpl_buffer(ctx)
24:     restart_pfm()
25:   break
26:   case PFM_MSG_END : /*monitored task terminated*/
27:     activeThreads--
28:     process_smpl_buffer(ctx)/*check leftover samples*/
29:   break
30: }
31: if ((activeThreads == 0) OR (monitoringPeriodExpired == TRUE)) then
32:     createPagelist(contextList)
33: end if
34: }
```

– An asynchronous notification is installed using a standard *fcntl* file control mechanism. When a SIGIO interrupt arrives, the `overflowHandler` function is executed.

– The *libpfm* library is initialised.

– Next, a child process is created. Its first action is to execute a `ptrace(PTRACE_TRACEME)` system call. This invocation provides a means whereby the parent process can observe and control the execution of the child. This action is followed by an `exec()` system call to start running the application to monitor.

– The code enters then in a loop in which it is notified every time a new thread is created by the child process. At the moment this happens, the thread will have previously been stopped by the `ptrace(PTRACE_TRACEME)` system call.

- A new `Perfmon` context is created and attached to the *tid* of the new thread. The context creation implies that a sampling buffer will be allocated and associated to that context. The buffer is configured to send a notification, via the associated file descriptor, every time it is full. In our experiments, the event to sample is "`DATA_EAR_CACHE_LAT4`", which corresponds to all cache misses with latency higher that 4 cycles.

- Finally, a `ptrace(PTRACE_ATTACH)` system call begins tracing the new thread before it is resumed to start running normally.

All this process allows detecting dynamically any new threads created or defunct during the monitored application's life cycle.

As previously explained in Section 5.2.2, the sampling buffer of each context is configured to send a notification every time it overflows. An interruption is raised and, then, the `overflowHandler` function handles it, following the steps depicted in the flow diagram of Figure 5.5:

- First, the *tid* of the thread that raises the interruption is obtained and the `Perfmon` context associated to it is accessed.

- `Perfmon` stores buffer notifications as messages in a queue. A standard `read` library call is used to read the message. If the message indicates that the buffer got full and overflowed (`PFM_MSG_OVFL`), the buffer is processed (subprocess *A*) and the context is restarted to get ready to keep on storing samples. If the message indicates that the thread has finished, then the leftover samples which could be still stored in the sampling buffer (not enough to make it overflow) are read, and that thread is stopped being traced by our monitoring process.

- When either the monitoring period expires or the monitored program finishes, a *page map* is created/updated with fresh information from the just read sampling buffer (subprocess *B*). Obviously, there is no point in creating a page map once the program has finished, but it is useful for debugging purposes and can be disabled in a production environment.

Flow diagram *A* in Figure 5.5 shows how the sampling buffer is processed each time a notification arrives to the monitoring tool:

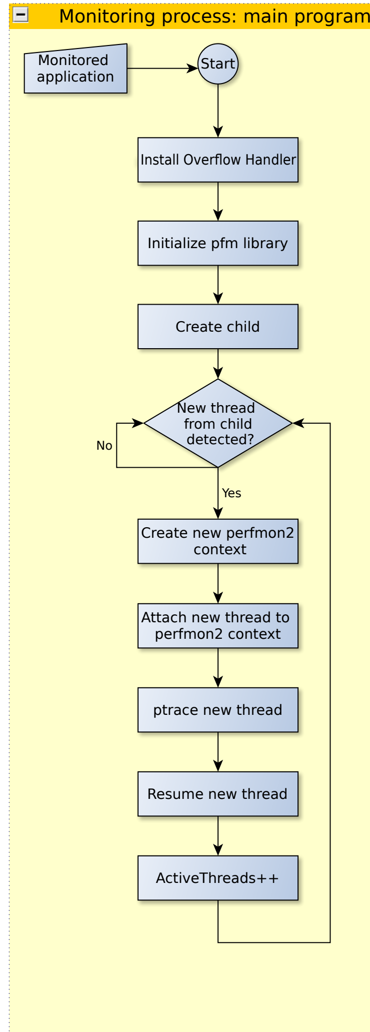



**Figure 5.4:** Flow diagram from the main function of the monitoring tool.

– Each `Perfmon` context has a pointer to its associated sampling buffer's header address. The number of stored samples in the buffer is obtained from the buffer header, so that the samples can be iterated.

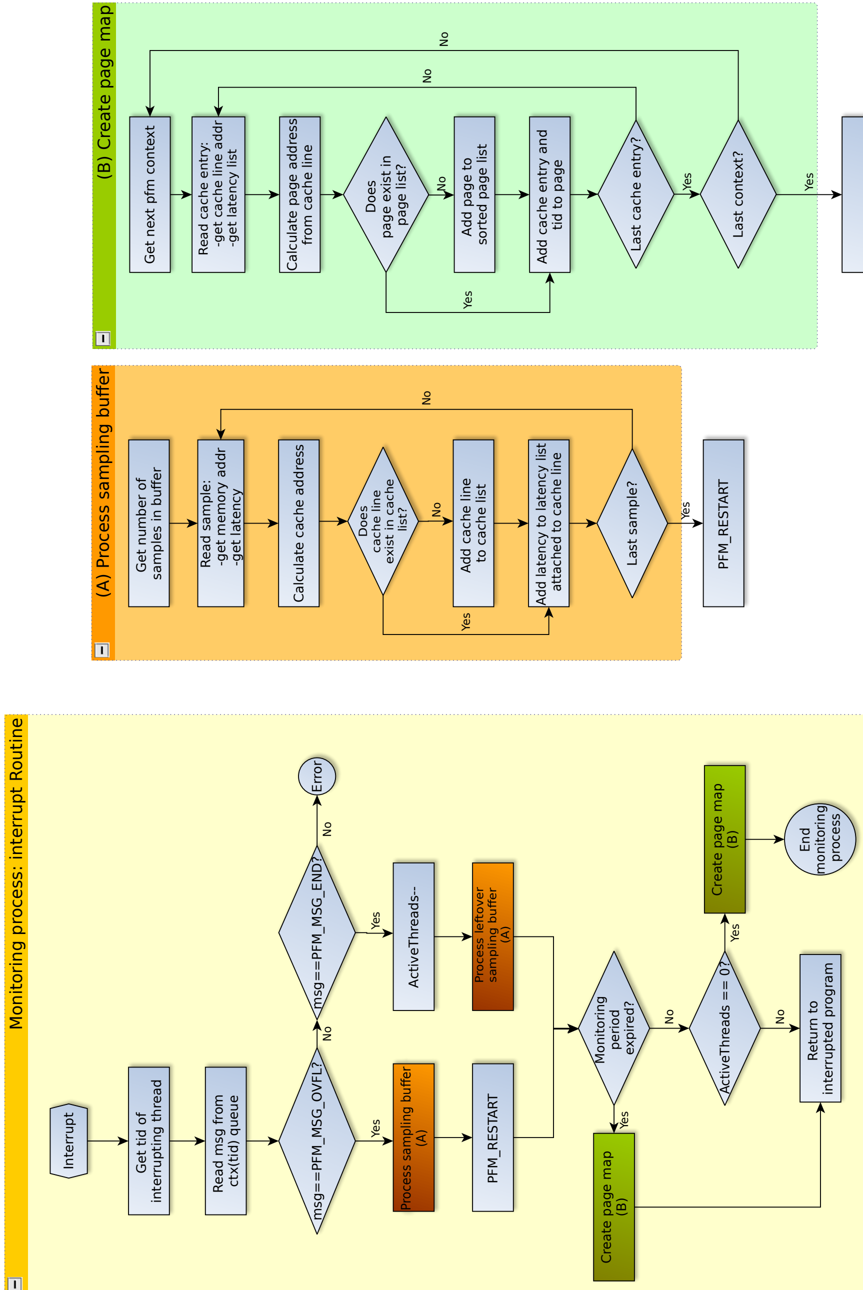



**Figure 5.5:** Flow diagram from the interruption handler of the monitoring tool.

- For each sample, the memory address, as well as its associated latency, are retrieved.
- Taking into account L2 and L3 cache sizes (note that L1 is ignored for real types) the cache line is identified from each address.
- Cache lines are stored in a dynamic linked list associated to the context. Note that, therefore, there will exist one linked list per monitored thread to keep control of each thread accesses. If that line was already accessed it will exist in the list, so the new latency is just appended to that line. Otherwise, the new line and its latency are added to the list.
- This loop continues until all the samples in the buffer have been read.
- When sampling is restarted via `PFM_RESTART`, the sampling buffer is marked as empty.

Flow diagram *B* in Figure 5.5 shows how the page map list is created, once all cache lines have been accessed:

- The list of existing contexts is iterated. For each context, each cache line accessed is read.
- The page address to which each cache line belongs is identified and appended to the list (which is actually a binary search tree) associated to that context.
- If the page already exists in the list, the cache line is added to that page. Otherwise, a new page will be created.
- This process is repeated for all cache lines accessed per context and for each context.

The creation of the page map is the last step of the *monitoring stage*. Next, the *evaluation stage* begins. In this stage, the collected data is processed by a given migration algorithm and, then, the memory pages chosen by the algorithm are actually migrated. The migration algorithms developed are presented and discussed in Sections 5.5 and 5.6. Previously, a set of operating tests carried out to evaluate the performance of our page migration infrastructure are presented in the next section.

## 5.3 Operating tests

### 5.3.1 Access to page information

Our page migration infrastructure was tested with the OpenMP version of the parallel NAS benchmark suite v3.3 [83]. As an example, the `BT.B` (block tridiagonal, class B) benchmark, whose description can be found in [84] is presented here. It was executed on a FINISTERRAE rx7640 [57] node using four threads while being managed by our page migration infrastructure. This test was performed to check that the infrastructure works properly and provides useful information for the page migration problem.

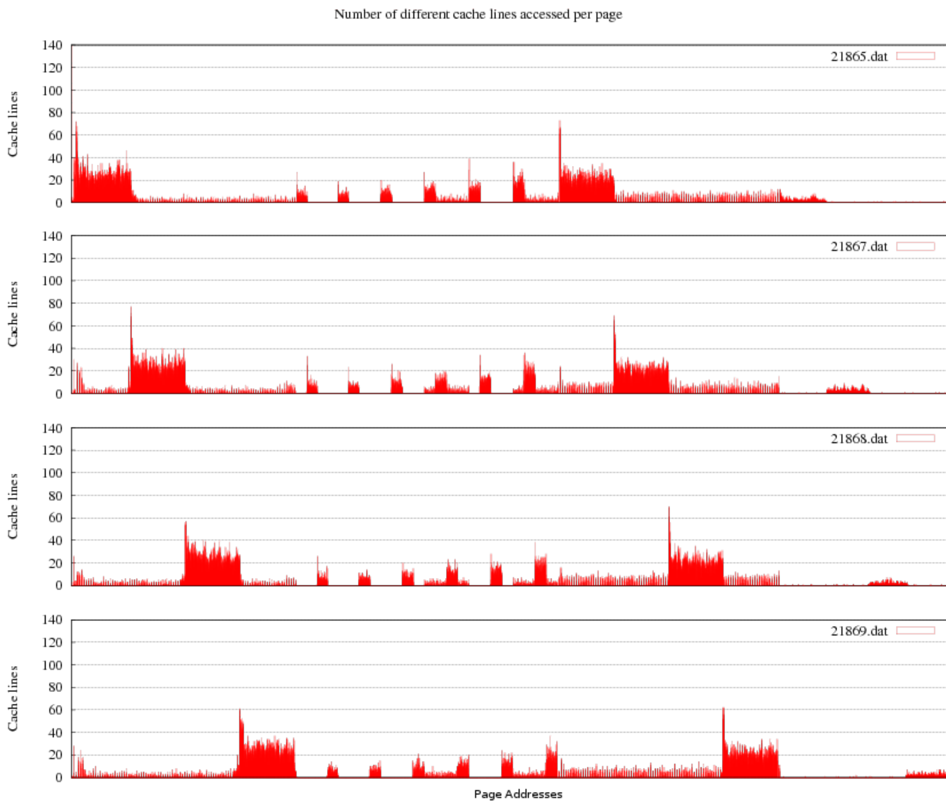



**Figure 5.6:** Number of different cache lines accessed per page and per thread. Only one access per cache line is shown.

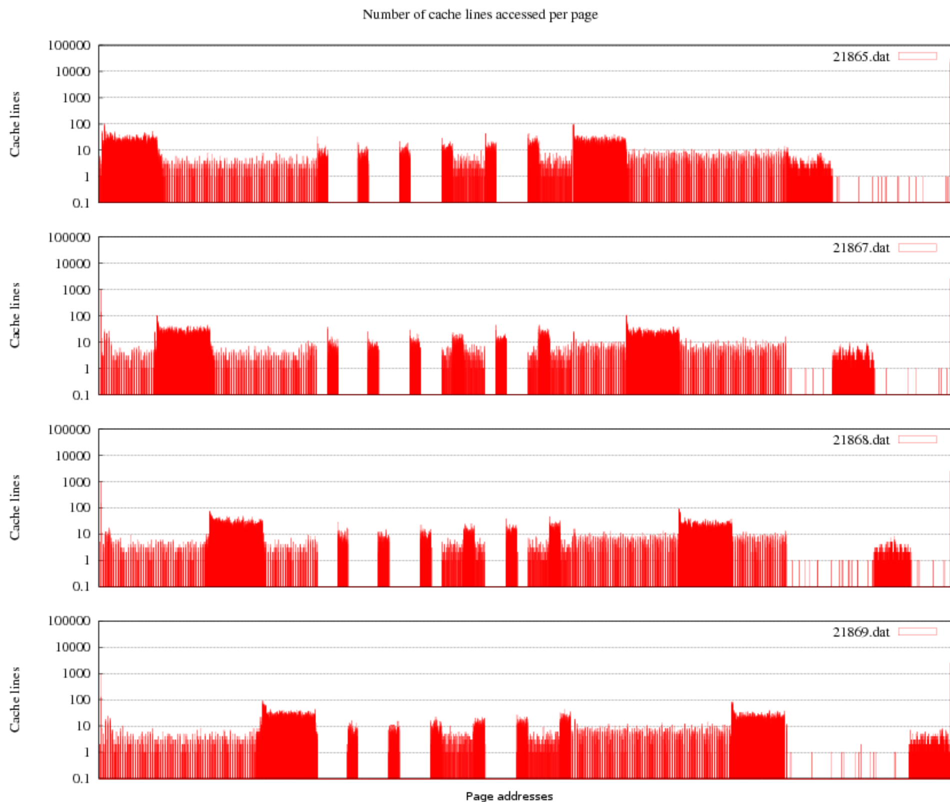



**Figure 5.7:** Total number of cache lines accessed per page and per thread. The sum of all accesses per cache line are shown. Y-axis in log scale.

Figures 5.6 to 5.9 show different outcomes for each thread. On each figure there is one graph per thread, identified by its `tid` number. On each graph the x-axis represents the whole range of sorted accessed pages. We observed that **icc** always creates an additional thread that, to our knowledge, performs no relevant operations for the computational task.

Figure 5.6 shows the number of different cache lines accessed per page by each thread. It helps to determine which pages had a larger range of addresses accessed. To know which pages had a higher number of total references, Figure 5.7 shows the whole number of references to a given page (note that Figures 5.7 and 5.8 are in logarithmic scale). It is noticeable that, due to the structure of the tridiagonal problem, there are several chunks of pages that are accessed mostly by a single thread and very little by the rest. This information about latencies

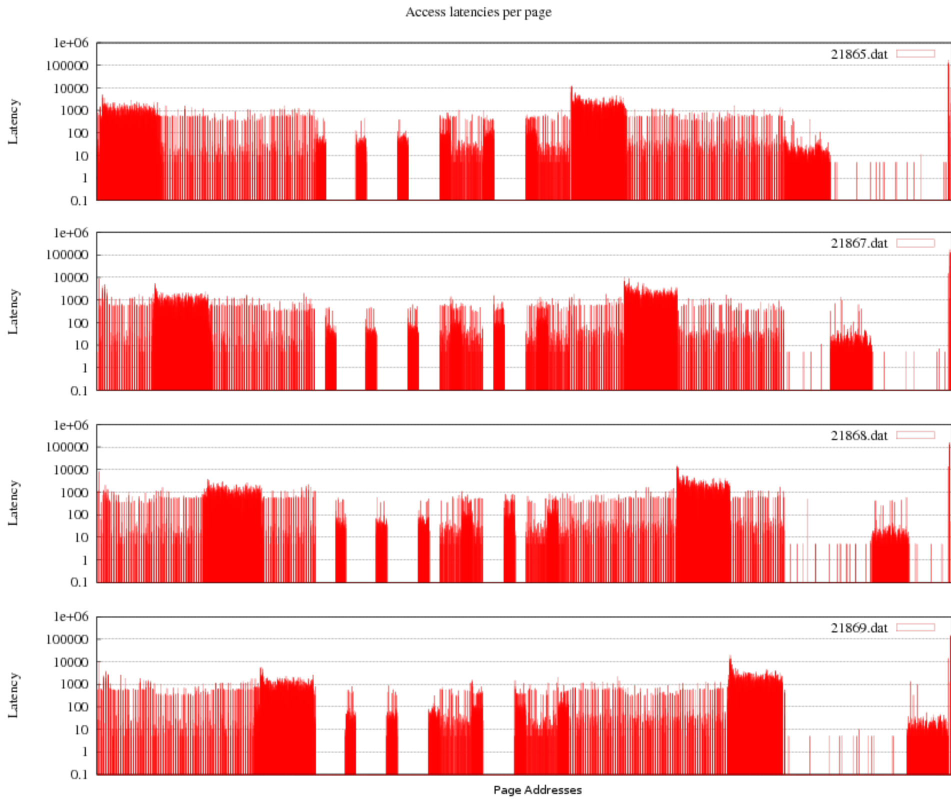



**Figure 5.8:** Sum of latencies of all references to a given page. Y-axis in log scale.

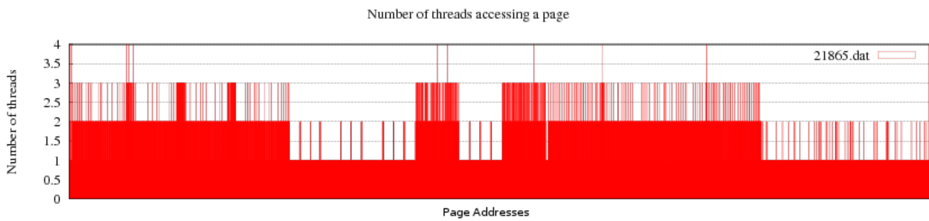



**Figure 5.9:** Number of threads accessing a given page.

can be correlated to the affinity set of each thread, thus allowing us to determine whether a page is local or remote to a cpu.

Figure 5.8 reflects the accumulated latency of all accesses to each page by each thread, which can be useful to take migration decisions. Finally, Figure 5.9 shows the number of threads that accessed a given page. This permits us to know which conflicts exist among which threads.

The results obtained show the usefulness of our development. It is possible, even at first sight, to observe the number of threads accessing a given page, the locality of data, the regions where any conflicts can occur and the distribution of pages among the threads.

### 5.3.2 Migration test

In order to test the behaviour of the *evaluation stage* of our page migration infrastructure and assess to what extent a good page migration policy can improve the performance, a proof of concept was developed. For that purpose, a small OpenMP benchmark, shown in Algorithm 4, was written. It comprises two steps. In the first step (lines 6 to 13), a master thread allocates a large array of doubles and then accesses it, to ensure that the data is locally allocated by the first-touch policy. In the second step (lines 15 to 22), each thread accesses an independent chunk of the array randomly, in order to force a noticeable number of different accessed pages. This benchmark was run in one of the FINISTERRAE rx7460 nodes using two threads. Note that an access is called *local* when it comes from a processor in the same cell as the involved memory module, and *remote* when it comes from another processor in a different cell.

The FINISTERRAE's thread scheduler behaves in a way that, although both threads had initially been allocated in the same cell, one of them was always migrated to the other cell. Since the data had previously been allocated in the master's cell memory by the first touch policy, the migrated thread was forced to access its dataset remotely. This is verified in Figure 5.10. It shows the latency of each access made by every thread on each cell. For instance, when a thread allocated in Cell 0 accesses data remotely, this cell (Figure 5.10(b)) will show a high latency value. In our benchmark, data are initially allocated by the master thread in Cell 1 (Figure 5.10(a)). All accesses from that node are, therefore, local, so the access latencies fall in the range of 300-350 cycles. Next, the benchmark migrates one of the threads to Cell 0 and, as shown in Figure 5.10(b), all accesses from there are remote so the latencies are in the range of 500-600 cycles.

**Algorithm 4:** Parallel benchmark to test local and remote accesses

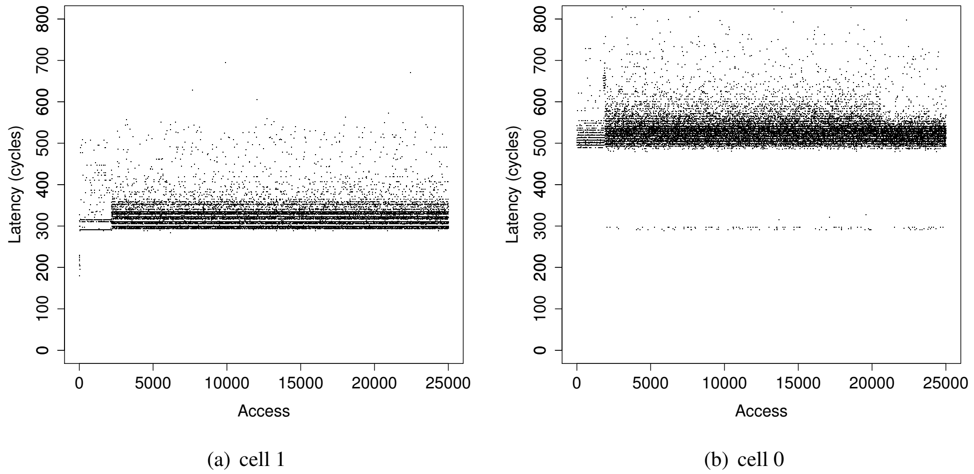
```
1:  do_loop()
2:  {
3:  #pragma omp parallel
4:  {
5:  thread ← omp_get_thread_num()
6:  /*1st step*/
7:  #omp_master
8:  {
9:  pageArray ← allocate_array()
10: for (i = 0; i < length(pageArray); i++) do
11:     pageArray[i] ← pageArray[random()%length(pageArray)]
12: end for
13: }
14: #pragma omp barrier
15: /*2nd step*/
16: chunkSize ← length(pageArrayChunk)
17: for (4 times) do
18:     for (i = 0; i < chunkSize; i++) do
19:         pageArray[i] ← pageArray[thread * chunkSize + random()%chunkSize]
20:     end for
21:     #omp_barrier
22: end for
23: }// end parallel region
24: }
```

This test was repeated using our page migration infrastructure. Figure 5.11(a) shows again local accesses from the threads allocated in Cell 1. However, in this case the pages that initially were accessed remotely are now migrated to the cell of the accessing thread (Cell 0, Figure 5.11(b)) so, from then on, accesses are local, with the consequent time reduction.

Table 5.1 shows the average execution time of each loop and the total execution time including initialisation times and overheads (hence, it is higher than the sum of the slowest thread of each loop). Note that the first loop is dedicated only to allocate data. Therefore, the time measured using our infrastructure is slightly higher than the no-migration case due to a small monitoring overhead. However, from the second loop on, the time of the thread in Cell 1 –which is migrated to Cell 0 – drastically drops thanks to the page migration. The whole speedup achieved is 16.7%. All these experiments were performed using an array of 30000 pages (∼458MB) to make sure data do not fit in cache memory.

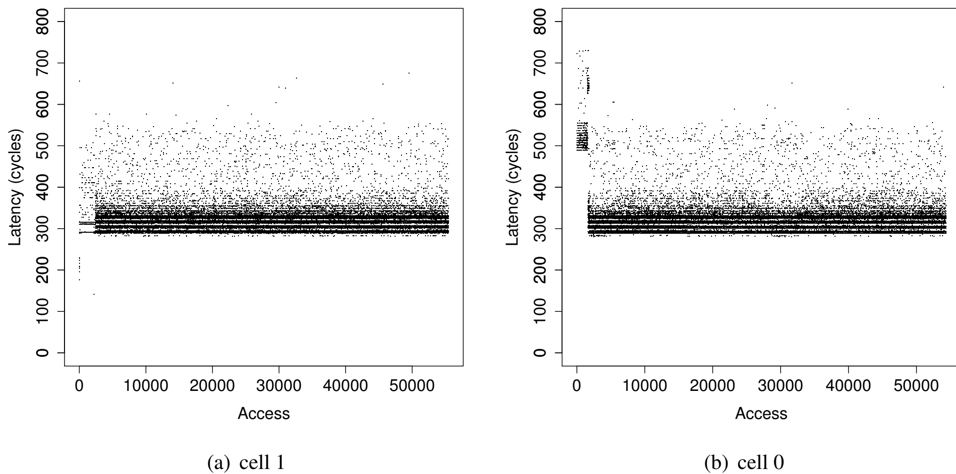
### 5.3.3 Migration performance study

Page migration is performed using the `move_pages()` system call [85]. Whereas this call permits page migration in user-space, it suffers from a limited performance. Indeed, each invocation to `move_pages()` has a large initialisation overhead and, among other side ef-




(a) cell 1 (b) cell 0

**Figure 5.10:** Access latency from threads on each cell. No migration.




(a) cell 1 (b) cell 0

**Figure 5.11:** Access latency from threads on each cell. Migration.

| Mode | Time(sec.) | | | Total | Speedup (%) |
|---|---|---|---|---|---|
| | Step 1 | Step 2 | | | |
| | Thread 0 | Thread 0 | Thread 1 | | |
| No migr. | 13.3 | 26.2 | 41.1 | 54.4 | |
| Migr. | 13.4 | 26.1 | 33.2 | 46.6 | 16.7 |

**Table 5.1:** Average execution time (sec.) of a data-intensive benchmark, with and without page migration.

fects, flushes the TLB. Although `move_pages()` was fixed in kernel 2.6.29 to have a linear complexity [47], its overhead to migrate large buffers might constrain the final performance of a page migration policy. In our infrastructure, the evaluation stage is run every time the monitoring stage ends. In this stage, a migration policy is applied to evaluate which of the sampled pages must be migrated. Then, they are actually migrated calling `move_pages()`. The larger the buffer of pages to migrate is, the higher the potential migration overhead. Hence, the overhead of `move_pages()` has been quantified to estimate whether page migration in user space is affordable in FINISTERRAE using our page migration infrastructure.

#### 5.3.3.1 Experiment setup

To estimate the performance of the `move_pages()` call, the following issues were taken into account:

- The study was carried out on a rx7640 node, shown in Figure 2.1, running a 2.6.29.6 Linux kernel. Both cells are symmetrical and connected through a crossbar. Considering that there are no other processes interferring, it seems reasonable to formulate as a hypothesis that the time to migrate a page from Cell 0 to Cell 1 must be the same as from Cell 1 to Cell 0.
- By default, page size on FINISTERRAE is set up to 16 KBytes. The performance of `move_pages()` is related to the number of pages to migrate. Therefore, the experiment must evaluate the performance related to the number of 16 KB pages.
- Our page migration infrastructure orchestrates the monitoring and migration process from a core different from the ones in which the monitored program runs. The basic migration policy implemented for this test states that, when the number of remote accesses to a page $p$ gets higher than the number of local accesses, $p$ will be migrated

from the local cell to the remote one. The experiment must evaluate whether there is any difference in performance ordering the migration from either a local or a remote cell.

The following section shows the migration performance of a code in C that allocates P pages in a buffer, places a thread in a given core and calls `move_pages()` from that thread. The throughput related to the time to complete the buffer migration from one node to another is measured.

#### 5.3.3.2 Migration throughput

Figure 5.12 shows the throughput of `move_pages()` for different number P of pages and migration strategies. The legend *"Data in cell X. Thread in cell Y. Migration from X to Z"* means that the data to migrate is initially allocated in cell $X$ and the thread that invokes `move_pages()` is in cell $Y$. The pages to migrate are in cell $X$ and are moved to cell $Z$.

The figure clearly shows two different trends in the four experiments. In two out of the four cases, in which the thread that invokes `move_pages()` is in the same cell as the data initially are, the throughput is identical to each other and higher than the opposite cases. This effect is specially noticeable for 32 pages (maximum use of the L1 DTLB). In all experiments, the peak throughput is achieved for 128 pages, which is the number of entries of the L2 DTLB. From then on, the performance falls until a buffer size of 2048 pages is reached, staying approximately steady from that value on.

In preliminary tests performed using our page migration infrastructure the number of pages to migrate has been checked to be practically always higher than 1024 pages per monitoring period (otherwise, to avoid unnecesary overheads, page migration could even be disabled). These values coincide with the steady region in Figure 5.12. The throughput in that region is in the range of 1150 MB/s for the best cases and 1115 MB/s for the worst ones. The difference is just about 3%. Furthermore, taking into account that in the tests the number of pages migrated on each monitoring period was in the range between 1024 and 16384 pages, the migration time can be assumed to fall, approximately, in the range between 14 and 229 milliseconds at most. It is not straightforward to state whether such an overhead is affordable or not, since it will depend on the number of page migration actions that occur during the execution of the monitored program and how much these and the remaining overheads are compensated with the improvement achieved by those migrations. Results in sections 5.5 and 5.6 show examples in which these situations take place.

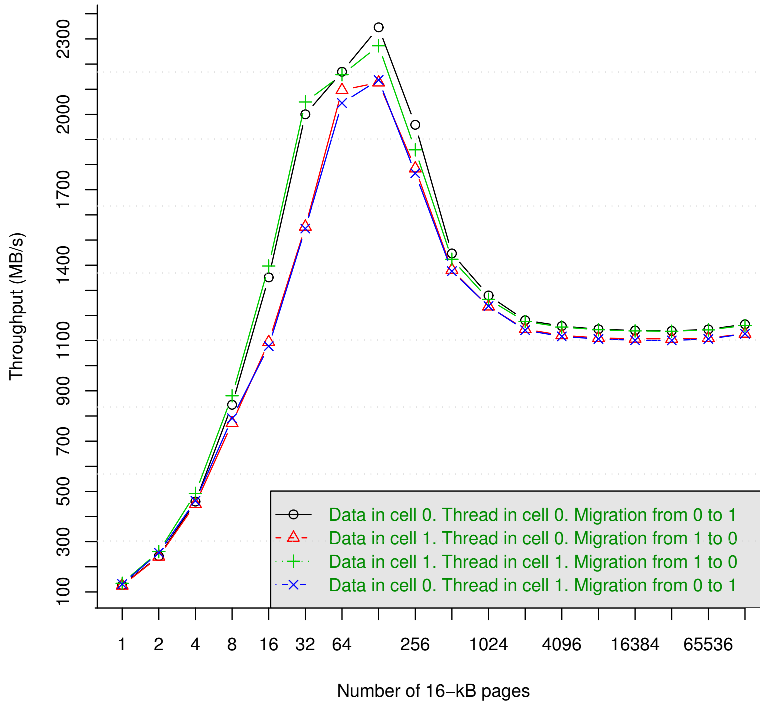
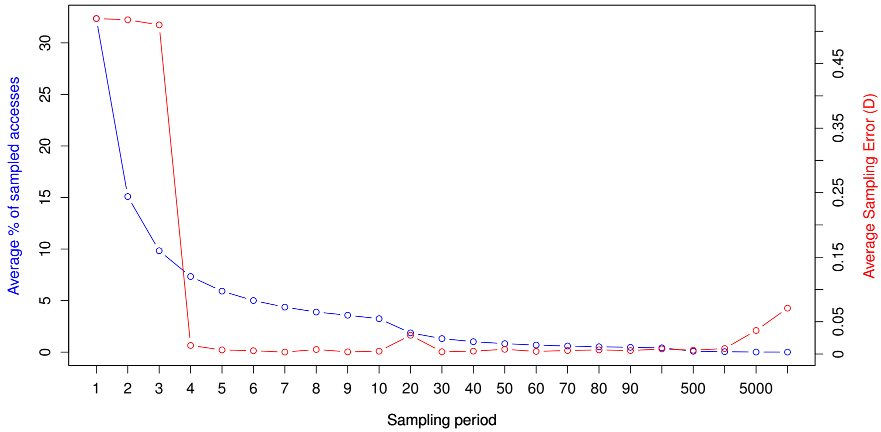


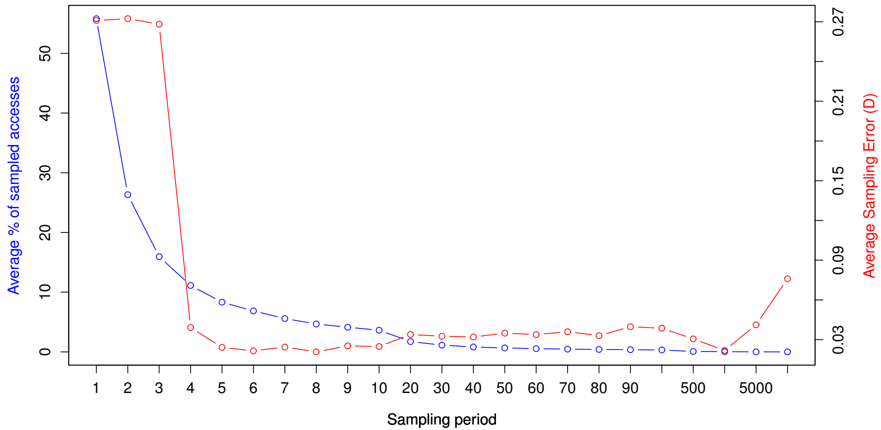

**Figure 5.12:** Migration throughput (MB/s) between Cells 0 and 1.

#### 5.3.3.3 Reliability of sampling

Our page migration framework comprises a sampling stage susceptible of introducing some overhead. In theory, the lower the sampling period, the higher the number of samples recorded. In practice this is not always true, since hardware counters have inherent physical limits. Indeed, a sampling period too low may result in many samples missed, since there is a certain waiting period between overflows before the monitoring process can be restarted. Conversely, a too high sampling period will not sample enough pages to obtain a set that is representative enough of the monitored workload. Hence, a range of sampling periods in which the sampled values are reliable and representative must be found.

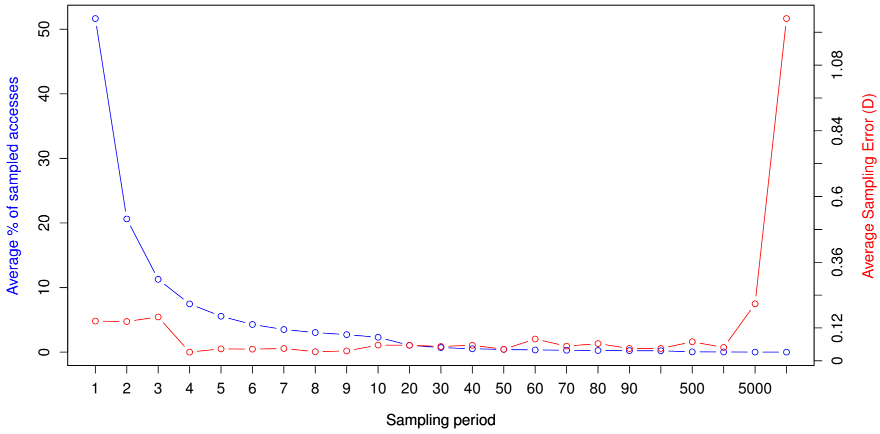



(a) 2 threads

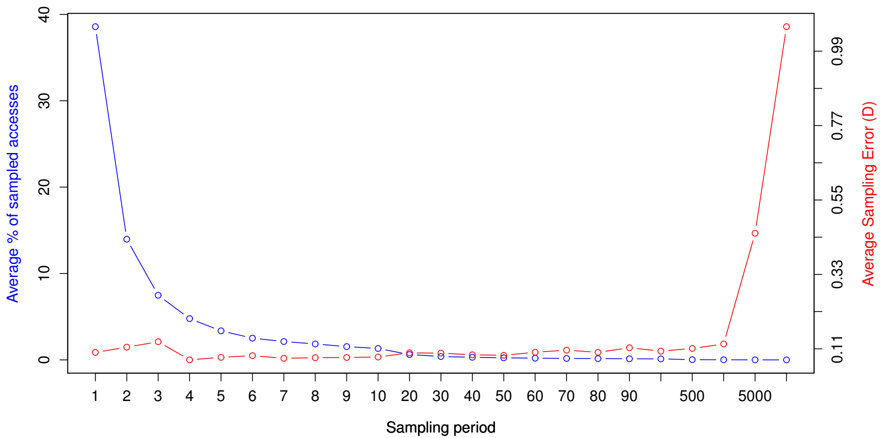



(b) 4 threads

**Figure 5.13:** Average distance for 2 and 4 threads




(a) 8 threads




(b) 16 threads

**Figure 5.14:** Average distance for 8 and 16 threads

A number of experiments were conducted to compare how representative our sampling method is of all memory accesses. A distance metric $D$, similar to the one described in [35], was used. If we consider the *ratio of accesses* from a processor to the total number of accesses, $D$ measures the difference, or error, between ratios of accesses for a given program in the following procedure:

- An OpenMP program which randomly accesses positions of an array is used. Variable $C_p$ counts the number of accesses by processor $p$. Another variable, $C_a$, counts all accesses by all processors.
- The OpenMP program is monitored by our page migration infrastructure, using EARs to sample the accesses of each processor to array $S_p$, and the accesses of all processors ($S_a$).
- The ratios of actual and sampled accesses by each processor are given by $R_{all} = \frac{C_p}{C_a}$ and $R_{sample} = \frac{S_p}{S_a}$, respectively.
- $D$ indicates how much a set of accesses deviate from another set and is defined as $D = \frac{|R_{sample} - R_{all}|}{R_{all}}$. Therefore, the closer the distance to 0 is, the more representative the set of sampled values is to the set of all accesses.

The experiment was performed for 2, 4, 8 and 16 threads on a rx7640 node. Results are shown in Figures 5.13 and 5.14. They combine the average distance of every processor (in red) with the percentage of sampled accesses (in blue) for every case. All figures show a similar pattern in which a low sampling period (1 to 5, approximately) obtains a high number of samples, but at the cost of a noticeable distance error. From a sampling period of 5, the values of $D$ decrease dramatically and stay steady until it increases again for values higher than 1000, due to the fact that the number of samples is low enough to not characterise accurately the monitored program. The choice of an appropriate sampling period must attend at a low value of $D$ and, simultaneously, a sufficient number of samples. In view of the figures, regarding $D$, an appropriate choice for any number of threads may be any sampling period between 5 and 1000. However, the choice regarding the percentage of sampled values is far from being trivial. Indeed, at first sight, a sensible range of sampling periods could be any between 6 and 10. These periods are enough to obtain between 5% and 10% of sampled values. However, this experiment evaluates a constant monitoring stage. The overhead associated to the evaluation stage, in which the sampled data are processed, is not considered.

Empirically, a period of 1000 has been verified to be enough to acquire a representative number of samples, so that was the sampling period used in the experiments presented in the remaining sections. The monitoring period chosen has been 1 second, which also proved to obtain the best ratio overhead-number of useful samples.

## 5.4 Affinity decisions

Section 5.3.2 showed an example in which the decision to migrate a page is driven by the number of local and remote accesses to that page. Despite being a straightforward technique, this page migration policy has proved to be effective to approach this problem. Indeed, several authors have used it successfully in the past. For example, in [35] it was used as a user-level strategy to migrate pages in a cc-NUMA UltraSPARC III, taking advantage of the hardware counters provided by that architecture. Another example is [47], in which a next-touch policy was implemented for the Linux kernel in i386 architecture. This development presents the drawback of having not been yet introduced in the mainstream kernel, so the kernel needs to be patched and recompiled to provide it with that feature.

To our knowledge, nobody has hitherto developed an efficient user-level migration policy for the Itanium architecture. Subsequent sections introduce the algorithms developed, relying on the framework presented in this chapter, to take advantage of some of the new features provided by the PMU of the Itanium2 Montvale. The algorithms must be adaptable to different scenarios which comprise different latency-level memory hierarchies. FINISTERRAE is composed of Superdome nodes (Figure 5.15) and rx7640 nodes (Figure 2.1). As shown in the figures, while a rx7640 presents only two levels of memory latency per cell (local and remote), a Superdome node has a more complex structure composed of four groups of four cells each. Therefore, each cell will see three latency levels: local, remote inside the same group, and remote when accessing another group. Considering these structures, two general-purpose algorithms are proposed next. Then, they are particularised and evaluated on a rx7460 node.

## 5.5 Access-based migration algorithm for N-cell nodes

Figure 5.16 shows a tree diagram of the FINISTERRAE's Superdome node (Figure 5.15). In this case, there are four groups of four cells each. Every group is interconnected to each other at the same distance, understood as the same latency access value.

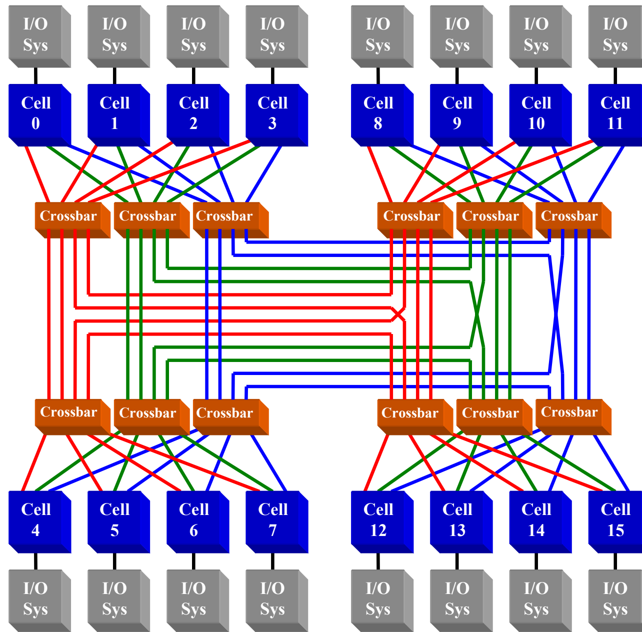



**Figure 5.15:** HP Superdome node

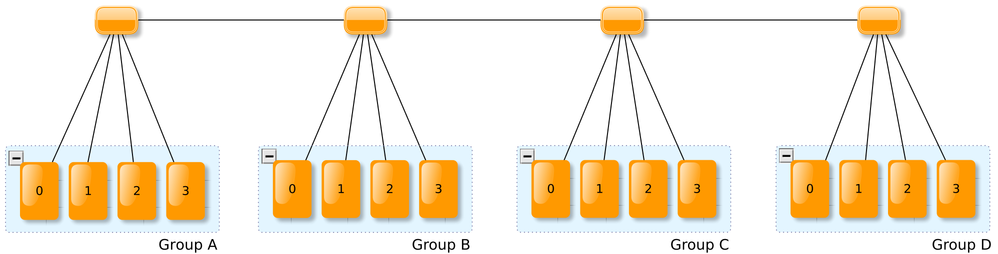



**Figure 5.16:** Superdome Node Graph

The access-based migration algorithm takes into account uniquely the number of times a page is accessed from each cell during a program run. Note that only incomplete information from the sampled information, provided by the page migration framework, is used. Assuming that our framework is able to sample a representative number of accesses from each page, our migration algorithm for a $c$-cells, $g$-groups node can be formally stated as follows:

– Let us suppose that our system is composed of a set of $|G|$ groups $G = \{G_1, G_2, \ldots G_{|G|}\}$. A group $G_j$ is composed of $|G_j|$ cells $G_j = \{C_{j1}, C_{j2}, \ldots\}$, with $1 \leq j \leq |G|$. In Figure 5.16, $|G| = 4$ and $|G_j| = 4, \forall j$.

– Let $P = \{p_1, p_2 \ldots p_p\}$ be the set of pages accessed by a program during its execution. Let us suppose that, during a monitoring period of our framework, the page $p_i$ resides in the local memory of cell $C_{ks} \in G_k$.

– Let $a_i^{jm} \geq 0$ be the total number of sampled accesses to $p_i$ from the threads running in cell $C_{jm} \in G_j$, being $1 \leq j \leq |G|$ and $1 \leq m \leq |G_j|$. The total number of accesses to $p_i$ from all the cells in group $G_j$ is then

$$\alpha_i^j = \sum_{m=0}^{|G_j|} a_i^{jm} \tag{5.4}$$

– Let $h$ and $d$ be the indexes such as

$$\begin{aligned} \alpha_i^h &= \max_{1 \leq j \leq |G|} \alpha_i^j \\ a_i^{hd} &= \max_{1 \leq m \leq |G_h|} a_i^{hm} \end{aligned} \tag{5.5}$$

– Then, the page $p_i$ is migrated to the cell $C_{hd} \in G_h$. Notice that if $h = k$ and $d = s$, $p_i$ is not migrated (because $p_i$ is in the memory of $C_{ks}$), and that if $h = k$ but $d \neq s$ an intra-group migration is carried out.

In other words, our framework evaluates the number of accesses to each page from each cell. After a monitoring period, each page is migrated to the cell which most accessed it under the assumption that, if a page has been accessed most from a cell during a monitoring period, it will keep on doing it in further periods.

## 5.6 Latency-based migration algorithm for N-cell nodes

### 5.6.1 Motivation

As it was stated in [35], a high bus traffic can compromise the performance of a program. Indeed, in our tests we observed that, in those cases in which the scheduler mapped many

threads to cores that share a bus, the execution time was higher than in those with a better distribution.

We posed the hypothesis of a situation in which the workload in a cell is so high that it is worth accessing pages remotely rather than migrating the pages to the local cell. In other words, if there are too many threads performing local accesses, the access latency might get higher than the access latency to a remote cell. In these cases, even in a situation in which the access-based migration algorithm of Section 5.5 would take the decision of migrating a page, a *latency-based* algorithm might prevent that page from being actually migrated.

The first step to confirm our hypothesis was to check that simultaneous accesses to memory increase the access latency. To do that, an OpenMP program that allocates an array and creates a given number of threads, each of which accesses a different part of the array, was written. After allocating a buffer of 763 MB (48828 pages), running the program on a rx7640 node with 1 thread yielded an average access latency of 318.3 cycles. For 4 threads, the latency was 350.4 cycles, approximately a 10% higher.

Next, the following scenario was configured:

– The OpenMP program was started on a rx7640 node while being managed by our page migration infrastructure.

– The OpenMP program allocated an array in Cell 0.

– 6 threads started accessing their own part of the array in order to generate an important traffic in the cell buses.

– At a given moment, one of the threads is manually migrated to Cell 1. After a monitoring period, our migration infrastructure moves the dataset of that thread to Cell 1.

– Finally, the thread is manually migrated back to Cell 0. Its dataset is subsequently moved back to Cell 0 by our migration infrastructure.

The average execution time was 235.1 seconds. The experiment was repeated but, in this case, after migrating the thread back to Cell 0, its dataset was kept in Cell 1. The average execution time then was 228.8 seconds.

This example illustrates the fact that a high access latency due to a heavily loaded bus, together with the unavoidable migration overhead, can decrease the performance of a program.

Subsequent sections introduce formally an algorithm that evaluates the current state of each cell and estimates the latency of a page before it is migrated, in order to decide whether it should be migrated or not.

### 5.6.2 Algorithm

Consider the nomenclature defined in Section 5.5. Let us consider again that the page $p_i$ resides in the local memory of cell $C_{ks} \in G_k$. Let $t_i^{hm}$ be the arithmetic mean of the access time (latencies) to $p_i$ from a cell $C_{hm} \in G_h$. For all the cells in $G_h$, the mean of the access time to $p_i$ is:

$$\bar{t}_i^h = \frac{\sum_{m=0}^{|G_h|} t_i^{hm} \cdot a_i^{hm}}{\sum_{m=0}^{|G_h|} a_i^{hm}} = \frac{\sum_{m=0}^{|G_h|} t_i^{hm} \cdot a_i^{hm}}{\alpha_i^h} \tag{5.6}$$

Let $P_l \subset P$ be the set of pages that resides in the memory of any cell of group $G_l$. The average latency of the accesses from any cell in $G_h$ to any page stored in $G_l$ is

$$\tau_l^h = \frac{\sum_{j|p_j \in P_l} \bar{t}_j^h \cdot \alpha_j^h}{\sum_{j|p_j \in P_l} \alpha_j^h}, 1 \leq l \leq |G|, 1 \leq h \leq |G| \tag{5.7}$$

Let us consider $\forall l, 1 \leq l \leq |G|$

$$\mathrm{T}_l = \max_{1 \leq h \leq |G|} \tau_l^h \tag{5.8}$$

$\mathrm{T}_l$ is the maximum latency (in average) to access the pages in group $G_l$. Let $q$ be the index such as:

$$\mathrm{T}_q = \min_{1 \leq l \leq |G|} \mathrm{T}_l \tag{5.9}$$

So, $G_q$ is the group with the minimum latency (in average) to access its pages. Our algorithm considers that if

$$\mathrm{T}_q < \min_{1 \leq h \leq |G|} \bar{t}_i^h \tag{5.10}$$

then the page $p_i$ should be migrated to group $G_q$.

Now, two different situations have to be considered:

(a) $q = k$, so a intra-group migration could be necessary. In this case, a similar procedure is carried out. Let $P_{km} \subset P_k$ be the set of pages that resides in the memory of $C_{km} \in G_k$. The average of the access time from a cell $C_{kn}$ to the pages in $P_{km}$ is,

$$\sigma_{km}^{kn} = \frac{\sum_{j|p_j \in P_{km}} t_j^{kn} \cdot a_j^{kn}}{\sum_{j|p_j \in P_{km}} a_j^{kn}} \tag{5.11}$$

And let $\Gamma_{km}$ be, $\forall m, 1 \leq m \leq |G_k|$

$$\Gamma_{km} = \max_{1 \leq n \leq |G_k|} a_i^n \cdot \sigma_{km}^{kn} \tag{5.12}$$

Now, two alternatives are considered to take the decision of migrating or not:

i) Let $d$ be the index such as:

$$\Gamma_{kd} = \min_{1 \leq m \leq |G_k|} \Gamma_{km} \tag{5.13}$$

Then, if $d \neq s$ the page $p_i$ is migrated from $C_{ks}$ to $C_{kd}$.

ii) Let $d$ be the index such as:

$$\Gamma_{kd} = \min_{\substack{1 \leq m \leq |G_k| \\ m \neq s}} \Gamma_{km} \tag{5.14}$$

Now, only if

$$\Gamma_{kd} < \max_{1 \leq h \leq |G_k|} a_i^h \cdot t_i^{kh} \tag{5.15}$$

the page $p_i$ is migrated to the cell $C_{kd}$.

(b) $q \neq k$. In this case, the page $p_i$ is migrated to the cell $C_{qd}$ being $d$ the index obtained by the following equation, in a similar way to which it is done in Equation 5.13:

$$\Gamma_{qd} = \min_{1 \leq m \leq |G_q|} \Gamma_{qm} \tag{5.16}$$

Then, the page $p_i$ is migrated from $C_{ks}$ to $C_{qd}$.

Note that there are no particularities why any page can take longer to be accessed than another one, with the exception of not being indexed in the TLB. Therefore, the only reason whereby a page shows a higher latency is that the workload –and, hence, the average access latency– had increased in its cell at the moment of being sampled. Assuming that such an increase will last for, at least, the next monitoring period, then it will be worth migrating the page to other cell in which the latency is lower.

To summarise, this algorithm performs an inter-group migration according to load-balancing criteria, moving a page to the group where it has been estimated that the access latency will be the lowest. Then, the process is repeated again to select a cell inside the chosen destination group. We find two cases now: if that group is the same in which the page currently is, the cell with the lowest estimated latency access is chosen by comparing the actual measured access latency to that page with the estimated latencies in the rest of cells. However, if the chosen destination group is different from the one that has the page to migrate, then there will not be any previous measurements of actual access latencies to that page inside the cells of that group. In that case, only estimated latencies will be used to decide the cell where to place the page.

## 5.7 Evaluation in a dedicated environment

### 5.7.1 Experiment setup

A series of tests was conducted on a rx7640 node. The algorithms of Sections 5.5 and 5.6 were therefore particularised for $g = 1$ groups and $|G_j| = 2$ cells. Note that the total number of sampled accesses to a page $p_i$ from a given cell, defined as $a_i^{jm}$ in the previous equations, are provided by the EAR hardware counters.

Eight out of the nine OpenMP NAS parallel benchmarks [84] v3.3 were used. Namely, BT, CG, FT, IS, LU, MG, SP and UA. EP was not evaluated given that it does not have significant sharing of data.

The NAS benchmarks come in four flavours, $A$ to $D$, being $A$ the smallest problem size and $D$ the largest. The chosen size was $C$, since that suite can be executed in our system in a reasonable time to take enough samples to have statistically representative results. At the same time, with this size they run for long enough to give room to improvement using the developed dynamic page migration strategies.

The goal of the experiment was to compare the execution time of each benchmark with and without the page migration algorithms. The experiment was setup as follows:

- Each benchmark was initially run 30 times on a rx7640 node with 4, 8, 12 and 15 threads without any specific constraints, allowing therefore the kernel scheduler to allocate and migrate the threads on any core. The average of their execution wallclock time was calculated.

- Afterwards, each benchmark was run 30 times with 4, 8, 12 and 15 threads using our migration infrastructure and each of the page migration algorithms developed. The monitoring thread was collocated on a separate 16th core.

A clarification is required about the latency-based algorithm (Section 5.6). There is no way to study inter-group migrations given that there is only one two-cell node available with just two latency levels. Regarding intra-group migrations, two approaches have been considered: Firstly, the one explained in the algorithm, in which the actual measured latencies to the page candidate to be migrated are compared with the estimated average latencies if the page is migrated. Secondly, only comparisons between estimated average latencies. In this way, both approaches can be compared to study the influence of using the actual latencies to a given page instead of the average load measured in the cell.

Note that the benchmarks are executed on a dedicated environment in which a thread is expected to be collocated in a free core. That means that few or no thread migrations or preemptions are expected. Since the benchmarks are already optimised to be parallel-efficient, the first-touch policy of the system should be sufficient to execute efficiently the benchmarks. Therefore, only little improvement is expected using a dynamic migration policy.

### 5.7.2 Experimental results and discussion

Table 5.2 shows the wallclock execution time of each benchmark for the original, first-touch allocated case and each of the page migration techniques developed. The speedups of each technique are also provided. In the table, *Nacc* refers to the access-based algorithm, $Lat_{real}$ to the latency-based algorithm using actual latencies of accessing a page, and $Lat_{avg}$ to the second approach, using just average estimated latencies.

The results show that the dynamic migration techniques proposed obtain some improvement in a few cases but also slightly slow down the execution of some of them.

| *Benchmark* | *Time(sec.)* | | | | *Speedup (%)* | | |
|---|---|---|---|---|---|---|---|
| | 1*st touch* | *Nacc* | $Lat_{real}$ | $Lat_{avg}$ | *Nacc* | $Lat_{real}$ | $Lat_{avg}$ |
| *bt.C* | | | | | | | |
| 4 *threads* | 551.3 | 536.8 | 534.2 | 534.7 | **2.7** | **3.2** | **3.1** |
| 8 *threads* | 282.8 | 295.1 | 296.2 | 292.7 | -4.2 | -4.5 | -3.4 |
| 12 *threads* | 207.5 | 211.6 | 212.9 | 210.3 | -1.9 | -2.5 | -1.3 |
| 15 *threads* | 177.4 | 176.1 | 173.8 | 177.1 | **0.7** | **2.1** | **0.2** |
| *cg.C* | | | | | | | |
| 4 *threads* | 83.4 | 93.2 | 99.3 | 94.6 | -10.5 | -16.0 | -11.8 |
| 8 *threads* | 64.1 | 83.3 | 79.9 | 79.3 | -23.0 | -19.7 | -19.1 |
| 12 *threads* | 52.4 | 65.9 | 65.1 | 64.2 | -20.5 | -19.5 | -18.4 |
| 15 *threads* | 43.5 | 57.1 | 57.1 | 59.7 | -23.8 | -23.8 | -27.1 |
| *ft.C* | | | | | | | |
| 4 *threads* | 88.2 | 94.0 | 89.8 | 92.8 | -6.1 | -1.7 | -5.0 |
| 8 *threads* | 48.8 | 56.8 | 56.8 | 54.9 | -14.0 | -14.2 | -11.1 |
| 12 *threads* | 37.8 | 37.8 | 38.4 | 40.3 | -0.2 | -1.6 | -6.2 |
| 15 *threads* | 33.1 | 32.8 | 32.8 | 34.5 | **1.1** | **1.0** | -3.8 |
| *is.C* | | | | | | | |
| 4 *threads* | 553.8 | 560.4 | 559.8 | 558.5 | -1.2 | -1.1 | -0.8 |
| 8 *threads* | 288.2 | 288.1 | 291.7 | 290.3 | 0.0 | -1.2 | -0.7 |
| 12 *threads* | 194.4 | 196.5 | 199.6 | 203.0 | -1.1 | -2.6 | -4.2 |
| 15 *threads* | 162.8 | 162.8 | 160.8 | 160.1 | 0.0 | **1.2** | **1.7** |
| *lu.C* | | | | | | | |
| 4 *threads* | 451.2 | 443.6 | 407.1 | 430.0 | **1.7** | **10.8** | **4.9** |
| 8 *threads* | 204.9 | 214.0 | 208.0 | 211.0 | -4.3 | -1.5 | -2.9 |
| 12 *threads* | 152.6 | 156.5 | 155.9 | 154.6 | -2.5 | -2.1 | -1.3 |
| 15 *threads* | 149.7 | 132.7 | 141.2 | 135.8 | **12.8** | **6.0** | **10.2** |
| *mg.C* | | | | | | | |
| 4 *threads* | 41.3 | 36.8 | 39.3 | 39.0 | **12.1** | **5.2** | **6.0** |
| 8 *threads* | 30.8 | 30.4 | 30.8 | 32.1 | **1.1** | -0.3 | -4.2 |
| 12 *threads* | 24.8 | 24.2 | 24.1 | 24.3 | **2.3** | **3.0** | **2.2** |
| 15 *threads* | 20.7 | 21.4 | 20.5 | 21.0 | -3.3 | **0.8** | -1.3 |
| *sp.C* | | | | | | | |
| 4 *threads* | 623.9 | 624.7 | 637.8 | 616.4 | -0.1 | -2.2 | **1.2** |
| 8 *threads* | 384.3 | 399.9 | 396.1 | 382.1 | -3.9 | -3.0 | **0.6** |
| 12 *threads* | 285.6 | 283.8 | 287.3 | 282.4 | **0.6** | -0.6 | **1.1** |
| 15 *threads* | 244.6 | 238.3 | 241.5 | 238.2 | **2.7** | **1.3** | **2.7** |
| *ua.C* | | | | | | | |
| 4 *threads* | 1608.7 | 1609.0 | 1602.0 | 1603.7 | 0.0 | **0.4** | **0.3** |
| 8 *threads* | 910.7 | 923.6 | 924.4 | 921.5 | -1.4 | -1.5 | -1.2 |
| 12 *threads* | 651.8 | 655.4 | 656.2 | 656.1 | -0.6 | -0.7 | -0.7 |
| 15 *threads* | 552.2 | 558.0 | 551.3 | 548.7 | -1.0 | **0.2** | **0.7** |

**Table 5.2:** Performance of the OpenMP NAS suite using the default first-touch policy and the page migration strategies proposed in a dedicated environment.

BT, SP, LU and UA are simulated CFD applications that reproduce much of the data movement and computation found in full *Computational Fluid Dynamics* (CFD) codes. Most of the improvements are found in these benchmarks for 15 threads, in which the amount of data transferred is bound to be important and where the chances of having threads not being optimally collocated by the scheduler are high. In these cases, the *access-based* algorithm will push pages close to the threads that access them most, and the *latency-based* algorithm will move data to the least loaded cell.

The remaining benchmarks (CG, FT, IS and MG) are kernels that mimic the computational core of four numerical methods used by CFD applications. Little or no improvement is found here. A twofold trend was noticed in our observations. We observed that the scheduler does not follow a fixed policy to map threads to cores. For example, for 4 and 8 threads, some executions of the same benchmark had their threads equally collocated in cores of both cells and some had them collocated in the same cell. Hence, only those executions whose threads were spread out in both cells could benefit of our page migration strategies to improve the performance and overcome the implicit overhead of the monitoring stage.

CG and FT are benchmarks in which the performance slow down is particularly noticeable. The former calculates a conjugate gradient, showing little movement of data and an important number of L1/L2 cache misses, due to the irregularity of its operation with sparse matrices [84]. Likewise, FT is a computational-bound kernel that calculates a Fourier Transform using the FFT-based spectral method, with little data movement. In view of the results, in those cases with such high data locality the use of a migration policy is detrimental to the performance.

## 5.8 Evaluation in a multiprogrammed environment

### 5.8.1 Experiment setup

The results presented in the previous sections assumed that the target machine is exclusively available to the program tested. However, this is not a realistic scenario. Many applications that run in a supercomputer share the hardware resources with other programs. Threads may be preempted or migrated at any moment by the OS scheduler for load balancing purposes. Hence, the chances that a thread is migrated far from its dataset are higher than in a dedicated scenario. Additionally, the presence of numerous programs accessing memory

simultaneously will cause a higher bus load. In this situations, our page migration strategies are expected to achieve a better performance.

In this experiment, a parallel program called *NomadicNoise* has been used to simulate, in a controlled way, another program in a multiprogrammed environment and the tests of Section 5.7 have been repeated. *NomadicNoise* allocates an array locally and accesses it for a given period. After the period expires, the array is freed and the program migrates its threads to the opposite cell, resuming the process. For our tests, *NomadicNoise* was executed using 7 threads, a 10 GB array and a 15-second period. The interference of this program will force some threads of the NAS benchmarks to lose their affinity, being moved to other cells by the OS scheduler when there are free cores (4 and 8 threads), or preempted when not (12 and 15 threads). As in previous tests, there is no binding of threads to processors or any other intervention that alters the behaviour of the scheduler except for the monitoring thread, which is mapped to a core different from the set of cores used by the NAS benchmarks.

Note that the latency algorithm was developed based on the premise that accessing a page remotely located can be worthy when a high load in the local cell is observed. The scenario proposed, in which several programs share the hardware resources, is likely to increase the average latency time in a cell, expecting therefore to benefit from the latency algorithm.

### 5.8.2 Experimental results and discussion

Table 5.3 shows the wallclock execution time and the speedup of each benchmark for the original case and each of the page migration techniques in the above multiprogrammed environment.

The results of Table 5.3 show a general performance improvement in a multiprogrammed environment over the native first-touch policy.

The simulated CFD applications (BT, LU, SP and UA), which have large data movement, were the most benefited by the migration policies. As in the dedicated environment, FT and CG were slightly slowed down. However, the performance decrease was noticeably lower than in the previous case.

The results also confirm a better behaviour when the OS scheduler can migrate threads to free cores in the presence of the interfering program rather than those cases in which the threads are preempted. Indeed, the most benefited benchmarks get a higher improvement for 4 and 8 threads than for 12 and 15. For example, LU shows about 30% improvement for 4 threads, whereas there is no improvement for 15 threads.

| *Benchmark* | *Time(sec.)* | | | | *Speedup* (%) | | |
|---|---|---|---|---|---|---|---|
| | 1*st touch* | *Nacc* | $Lat_{real}$ | $Lat_{avg}$ | *Nacc* | $Lat_{real}$ | $Lat_{avg}$ |
| *bt.C* | | | | | | | |
| 4 *threads* | 632.0 | 619.7 | 617.0 | 619.2 | **2.0** | **2.4** | **2.1** |
| 8 *threads* | 358.8 | 338.3 | 338.8 | 338.6 | **6.0** | **5.9** | **5.9** |
| 12 *threads* | 348.2 | 345.5 | 344.2 | 344.3 | **0.8** | **1.2** | **1.1** |
| 15 *threads* | 356.4 | 352.1 | 351.8 | 352.6 | **1.2** | **1.3** | **1.1** |
| *cg.C* | | | | | | | |
| 4 *threads* | 112.7 | 114.7 | 120.2 | 125.0 | -1.8 | -6.2 | -9.9 |
| 8 *threads* | 86.0 | 87.0 | 91.3 | 89.8 | -1.2 | -5.8 | -4.2 |
| 12 *threads* | 105.0 | 110.4 | 112.5 | 115.6 | -4.9 | -6.7 | -9.2 |
| 15 *threads* | 154.6 | 154.1 | 152.9 | 155.5 | **0.3** | **1.1** | -0.6 |
| *ft.C* | | | | | | | |
| 4 *threads* | 98.4 | 103.5 | 104.7 | 102.7 | -4.9 | -6.0 | -4.2 |
| 8 *threads* | 58.4 | 61.0 | 59.7 | 59.8 | -4.2 | -2.2 | -2.3 |
| 12 *threads* | 53.6 | 55.4 | 59.3 | 58.0 | -3.3 | -9.6 | -7.5 |
| 15 *threads* | 56.0 | 56.7 | 58.8 | 56.1 | -1.3 | -4.8 | -0.2 |
| *is.C* | | | | | | | |
| 4 *threads* | 606.4 | 596.3 | 602.6 | 595.8 | **1.7** | **0.6** | **1.8** |
| 8 *threads* | 319.8 | 325.5 | 315.2 | 327.4 | -1.8 | **1.5** | -2.3 |
| 12 *threads* | 315.8 | 312.3 | 312.7 | 316.7 | **1.1** | **1.0** | -0.3 |
| 15 *threads* | 326.5 | 327.3 | 326.7 | 321.1 | -0.3 | -0.1 | **1.7** |
| *lu.C* | | | | | | | |
| 4 *threads* | 550.8 | 432.3 | 423.6 | 432.0 | **27.4** | **30.0** | **27.5** |
| 8 *threads* | 260.4 | 248.8 | 241.1 | 246.9 | **4.7** | **5.4** | **5.5** |
| 12 *threads* | 478.7 | 459.1 | 459.3 | 459.0 | **4.3** | **4.2** | **4.3** |
| 15 *threads* | 1366.4 | 1447.0 | 1476.0 | 1461.2 | -5.6 | -7.4 | -6.5 |
| *mg.C* | | | | | | | |
| 4 *threads* | 52.2 | 51.1 | 54.8 | 51.4 | **2.1** | -4.8 | **1.5** |
| 8 *threads* | 37.9 | 37.1 | 36.8 | 36.6 | **2.3** | **3.1** | **3.5** |
| 12 *threads* | 41.6 | 38.7 | 38.1 | 36.9 | **7.6** | **9.2** | **12.8** |
| 15 *threads* | 44.6 | 41.5 | 40.6 | 40.3 | **7.4** | **9.7** | **10.7** |
| *sp.C* | | | | | | | |
| 4 *threads* | 865.6 | 779.3 | 783.1 | 783.7 | **11.1** | **10.5** | **10.4** |
| 8 *threads* | 516.8 | 470.5 | 463.7 | 467.6 | **9.8** | **11.4** | **10.5** |
| 12 *threads* | 524.5 | 486.3 | 483.2 | 487.3 | **7.8** | **8.5** | **7.6** |
| 15 *threads* | 557.7 | 523.4 | 525.2 | 524.5 | **6.6** | **6.2** | **6.3** |
| *ua.C* | | | | | | | |
| 4 *threads* | 1856.6 | 1799.7 | 1799.8 | 1799.6 | **3.2** | **3.2** | **3.2** |
| 8 *threads* | 1068.9 | 1036.1 | 1041.7 | 1042.4 | **3.2** | **2.6** | **2.5** |
| 12 *threads* | 1236.1 | 1208.3 | 1203.7 | 1206.3 | **2.3** | **2.7** | **2.5** |
| 15 *threads* | 1406.5 | 1392.2 | 1394.0 | 1393.3 | **1.0** | **0.9** | **0.9** |

**Table 5.3:** Performance of the OpenMP NAS suite using the default first-touch policy and the page migration strategies proposed in a multiprogrammed environment.

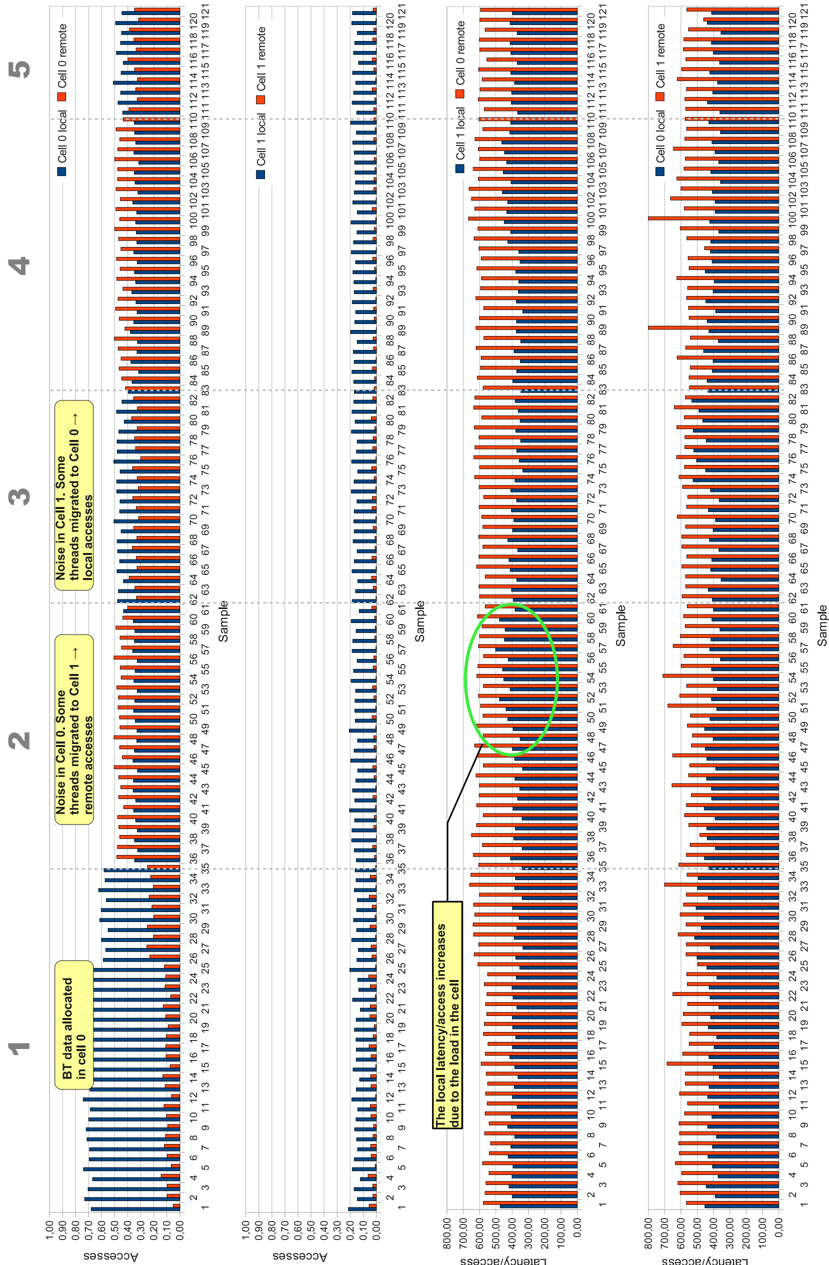



**Figure 5.17:** Number of accesses and latencies for BT. No page migrations.

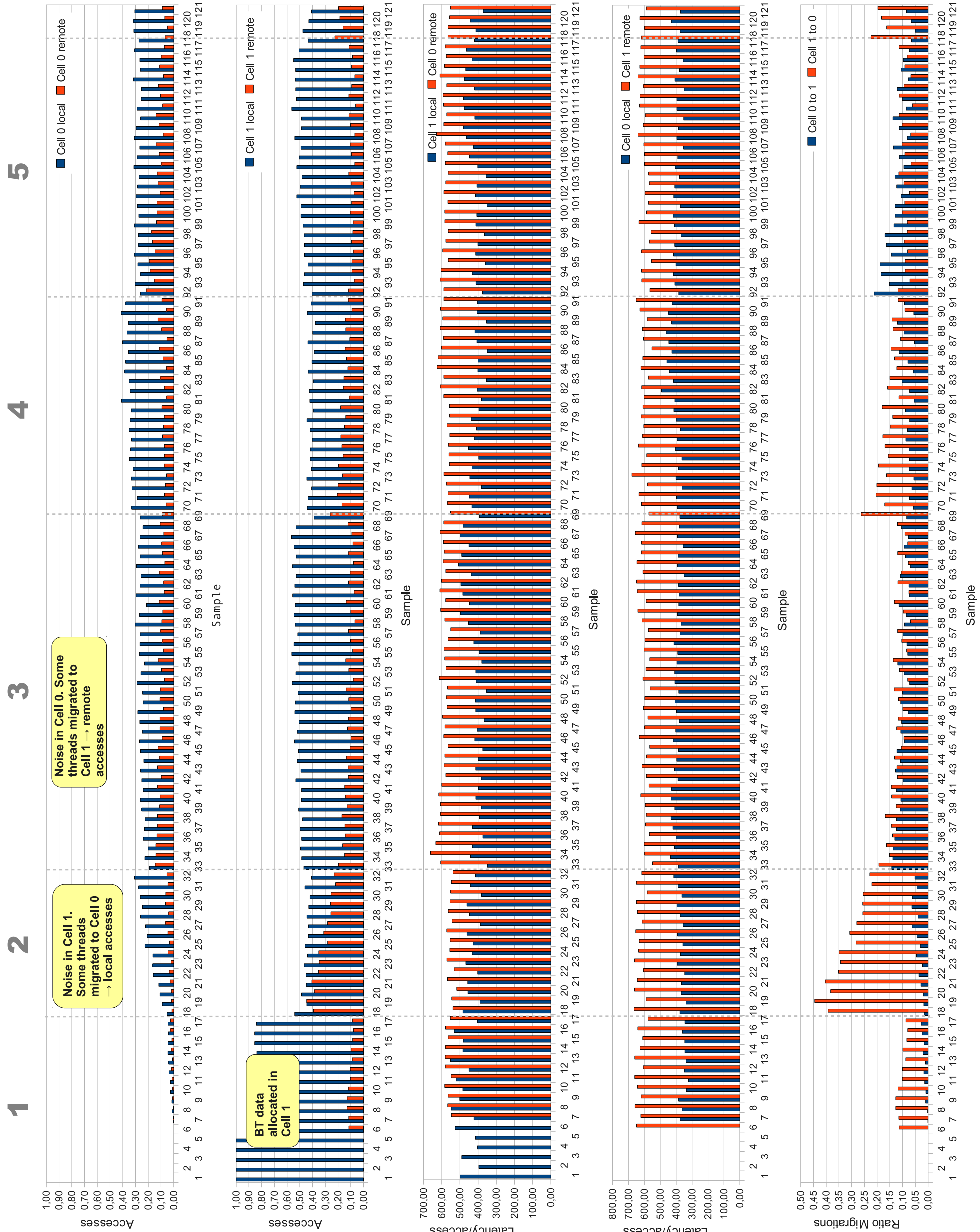



**Figure 5.18:** Number of accesses and latencies for BT. Access-based page migration.

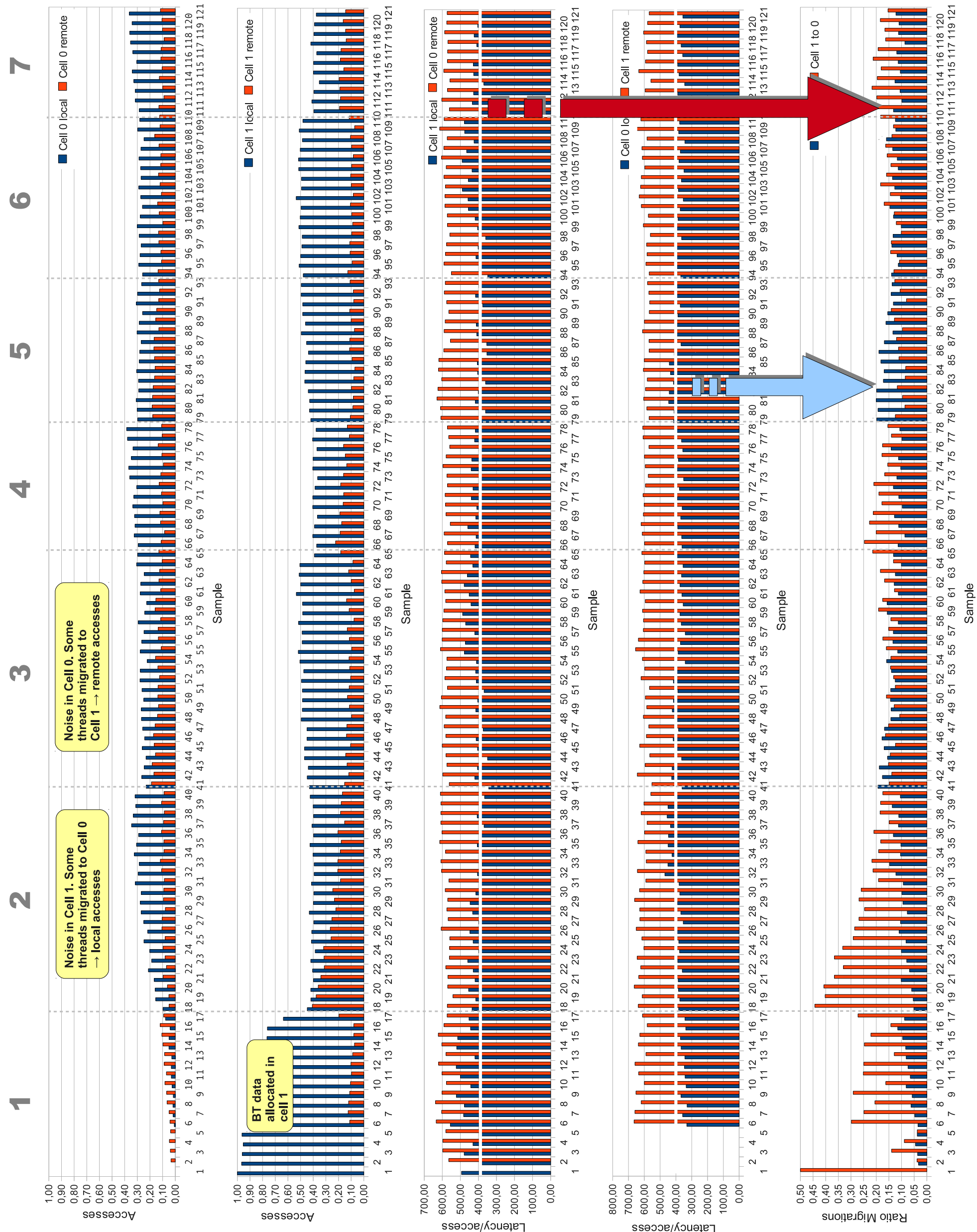



**Figure 5.19:** Number of accesses and latencies for BT. Latency-based ($Lat_{real}$) page migration.

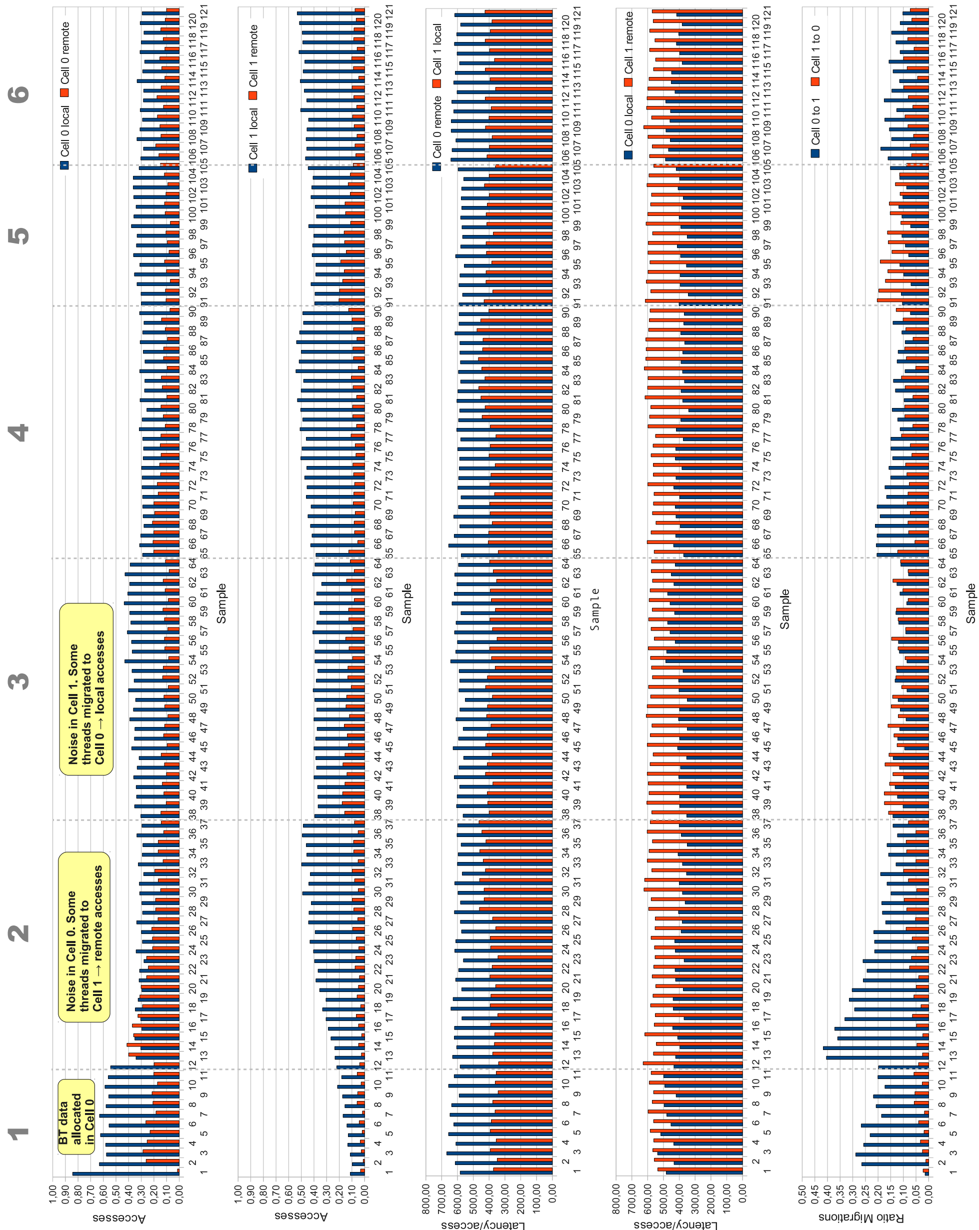



**Figure 5.20:** Number of accesses and latencies for BT. Latency-based ($Lat_{avg}$) page migration.

To understand better how our migration strategies behave, further analysis were carried out. The execution of BT is analysed in more detail next. Figure 5.17 shows the profile of a BT execution while is being interfered by *NomadicNoise*. Page migration is disabled. The figure comprises four histograms that represent certain magnitudes (*y axis*) measured for each monitoring period (*x axis)*. The grey dashed lines that cross all the histograms delimit the periods of *NomadicNoise*. After each period, it stops running in a cell and moves to the other one, forcing the OS scheduler to migrate the threads of BT to the opposite cell. This situation is reflected in the first two histograms. From up to down, the first one shows, from all accesses issued, the fraction of local (in blue) and remote (in red) memory accesses to Cell 0. The high number of local accesses to Cell 0 during the first period of *NomadicNoise* implies that BT has allocated its data locally[3]. Subsequent periods show an alternation in the type of dominating accesses, either local or remote. That is, the fraction of local accesses to Cell 0 are higher when *NomadicNoise* is in Cell 1, since it pushes the threads of BT to Cell 0 –where all their datasets are allocated–. Conversely, the remote accesses are higher when the interfering program is in Cell 0, since BT will access its datasets remotely.

The second histogram shows the fraction of accesses to Cell 1. The fraction of references issued is significantly lower (20% at most), since practically all the dataset of BT is allocated in Cell 0. These accesses are related to temporal calculations of the benchmark.

The third histogram displays the latency per access of the local accesses to Cell 1 (in blue) and remote accesses to Cell 0 (in red). By showing these values of local and remote cells together it is straightforward to compare how high the load is in each cell. For example, the green circle surrounds a zone where the local latency grows getting close to the remote latency. That situation occurs when the local cell is highly loaded, and accessing a page locally gets almost as costly as doing it remotely. Finally, the last histogram shows the latencies per access to Cell 0 and the remote ones to Cell 1.

Figure 5.18 depicts an execution of BT with page migration enabled using the access-based algorithm. The fifth histogram (bottom) shows the fraction of pages migrated from Cell 0 to 1 (in blue) and from Cell 1 to 0 (in red). In this algorithm, the decision to migrate a page is access-driven. Therefore, when the number of remote accesses increases in a cell, the page is migrated to the cell which most issued such data. For example, in the *NomadicNoise* period 2, BT has part or all of its threads migrated to Cell 0. Therefore, the number of remote accesses to Cell 1 increases up to a 45%, as shown in the second histogram (red bars). The

[3]BT allocates data statically using global arrays.

algorithm begins to migrate the accessed pages to Cell 1 until the number of remote accesses reduces to a minimum compared to the number of local accesses in Cell 0 (period 3 in the second histogram). The first two histograms show at a glance how an important number of remote memory accesses are eliminated when the algorithm performs the page migration.

Figure 5.19 shows an execution of BT using the latency-based ($Lat_{real}$ approach) algorithm. Now the migration decision is driven by the average latency of accessing a page, compared to the average access latency in each cell, for each monitoring period. A white horizontal line in the latency/access histograms shows the approximate average remote-to-local latency ratio. When the local access latency in a cell increases over that value, getting close to the remote latency to the opposite cell, chances are that issuing a local page in that period yields a high latency value. The algorithm may state that, considering that such load situation keeps for at least another monitoring period, having the page migrated remotely will reduce its access cost. Conversely, when the local access latency in a cell decreases over the average, the difference between accessing locally and remotely a page increases. If the page is allocated in another cell, then it will be worth pulling it locally.

The blue and red arrows in the figure illustrate two of these cases and how the migration algorithm takes a proper decision. The blue arrow shows a case in which the local latency in Cell 0 is increasing. This is correlated in the bottom histogram by an increase in the number of pages migrated whose accesses have taken too long and, therefore, are migrated from Cell 0 to Cell 1. The red arrow shows a situation in which the average local latency in Cell 1 increases, so the number of pages candidate to migrate from Cell 1 to 0 also rises.

An important fact reflected in these figures is that, once the average latency on each cell increases or decreases, it maintains this behaviour for several monitoring periods. Therefore, the migration is justified, since the cost of migrating a page is rapidly amortized.

Finally, BT executed with the latency-based algorithm ($Lat_{avg}$ approach) is shown in Figure 5.20. There are no significant differences with the previous approach.

By and large, the proposed techniques eliminate most of the remote accesses and increases the number of pages locally accessed. Table 5.4 shows the reduction in non-local memory accesses of the three experiments. Although the latency-based strategy ($Lat_{real}$) migrated a higher percentage of pages than the rest (28.7% vs 21.7% and 22.9%), the ratio of remote-to-local accesses is the highest (32.8%). This was due to an excessive ping-pong effect, in which some pages were often migrated from one cell to the opposite one in near or consecutive monitoring periods.

| | Pages | | Access locality (%) | | | | |
|---|---|---|---|---|---|---|---|
| | Sampled | Moved (%) | Cell 0 | | Cell 1 | | $\frac{\Sigma remote}{\Sigma local}$ |
| | | | Local | Remote | Local | Remote | |
| No migr. | 511092 | 0.0 | 16.7 | 2.1 | 41.1 | 40.2 | 73.2 |
| Nacc | 506613 | 21.7 | 30.0 | 9.5 | 48.8 | 12.1 | 27.4 |
| $Lat_{real}$ | 477308 | 28.7 | 27.9 | 12.2 | 47.3 | 12.5 | 32.8 |
| $Lat_{avg}$ | 511649 | 22.9 | 34.7 | 12.7 | 42.4 | 10.2 | 29.7 |

**Table 5.4:** Locality statistics of BT for each page migration strategy.

## 5.9 Conclusions

This chapter has introduced a profile-driven, page migration mechanism for Linux and the Itanium architecture. This mechanism relies on the *Event Address Registers* (EARs) available in the Itanium2 processors to obtain a sampled profile of the exact data addresses issued by a program. As a software contribution, a user-level monitoring and migrating tool has been developed. This tool attaches and samples the monitored program without need to modify whatsoever such a program. Moreover, it is flexible to support different monitoring and migrating strategies. After describing its architecture, a series of tests has been carried out to evaluate its performance and reliability, exploring the benefits and drawbacks of this approach.

Two page migration strategies for N-cell NUMA nodes have been proposed. A first access-based migration algorithm takes into consideration just the number of accesses sampled during a monitoring period in order to migrate a page to the cell that accessed it most. A second latency-based algorithm uses mainly the latency of the accesses, provided by the EARs, to state the load on each cell in runtime and migrate a page to the cell where its access cost is lower.

The effectiveness of these strategies has been evaluated using the OpenMP parallel NAS benchmarks on a two-cell rx7640 node. Two scenarios have been considered: first, a dedicated one, in which the main reason for non-local memory accesses is the first-touch policy in the operating system, should it had initially placed pages poorly in a cell. Depending on the parallelisation efficiency of each benchmark and the number of threads used, different results have been obtained. Data misplacing by the first-touch is more likely to happen for a high number of threads and, hence, there is more room for improvement. Best results were achieved for those benchmarks with large data movement, with a maximum execution speedup of 12.8%, although the general improvement was scarce and irregular.

The second scenario portrays a more realistic multiprogrammed environment in which another parallel program springs up periodically on each cell, forcing thread preemptions and migrations in the monitored program and, therefore, an important number of non-local memory accesses. In this case, the proposed migration techniques have been able to migrate pages according to their access pattern more efficiently than in the previous scenario, achieving a maximum speedup of 30%, in a more regular improvement trend.

Comparing the results obtained by each algorithm, the access-based algorithm seems to slightly overcome the rest. Although the differences among them are scarce, the competitive criterion of moving a page to the cell where it is requested more often comes up as the most effective in the rx7640 two-cell node. However, the latency-based approach is yet to prove its usefulness in NUMA nodes that comprise several-level latency modules, such as the Superdome node of FINISTERRAE. In this type of nodes this algorithm can act as a filter in order to avoid unnecessary page migrations or, conversely, foster others to lower-latency cells. Furthermore, despite the wide range of applications and kernels covered by the NAS benchmarks, the available space of data-intensive programs is still to be exhaustively explored.

# 6

# *Conclusions and Future Work*

Centred on a NUMA, Intel Itanium-based platform and its *Precise Event-Based Sampling* (PEBS) mechanism, this dissertation has presented novel techniques to improve the data locality and the performance of computational codes and applications in runtime. Specifically, two fields have been studied: reordering techniques for irregular codes and techniques for dynamic page placement. This chapter summarises the main contributions of the entire work accomplished throughout this thesis, presents the conclusions and proposes future lines of research.

The work developed can be summarised in the following points:

- **The architecture of the FINISTERRAE supercomputer has been evaluated.** In such a complex system in which each node comprises several cores interconnected by several levels of cache coherency in a NUMA distribution, a number of factors may affect the performance. We have found out that, in terms of cache coherency and performance, each core in every dual-core Itanium2 socket behaves as an independent processor. For data sizes larger than the cache size, the proper thread-to-core mapping is the one that keeps threads as far as possible from each other. This implies collocating threads in different cells when their datasets can be allocated locally, or in the same cell but in different buses when not. In this regard, **the Roofline Model has been developed for FINISTERRAE** and the Sparse Matrix-Vector product (SpMV) has been used as a case of study. This model has proved its usefulness to detect constraints in the performance of a kernel and to suggest possible optimisations upon thread and data allocations.
- **The access to the Itanium2 Montvale hardware counters has been studied.** After evaluating several tools to handle the processor's PMU, `Perfmon` was selected as the

most suitable interface for our requirements. Specifically designed in the beginning for the Itanium2 platform, it offers several features which have been essential in the development of this thesis, such as kernel-level sampling buffers and complete access to the information provided by EARs, the Itanium2 implementation of the PEBS registers. **Both dense and sparse codes have been evaluated.** The SpMV has been used as a case of study to test the behaviour of the sampling process. An average percentage of samples in the range of 10%-20% entries have proved to be sufficient for being representative of the sampled data.

– Regarding locality improvement on irregular codes, this work has centred in on using latency information provided by the EARs to improve the behaviour of data reordering techniques. One of their major problems is the reordering cost. **A particular technique aimed to improve the locality of sparse codes has been ported to FINISTERRAE and adapted to be used with sampled information from the hardware counters** in order to minimise the above drawback. This technique reorganises the data guided by a distance-based locality model. A study has been carried out to state the amount of information required to obtain similar results to the original technique but using sampled information. In this study, matrices consist of a subset of the nonzeros from the original ones. The nonzeros were randomly selected. The results showed that 1%-2% of sampled values is enough for Itanium2.

– **A novel method has been contributed to obtain the exact positions in a sampled matrix using hardware counters.** The same study was repeated using this method, finding similar performance to the original reordering technique. Speedups with respect to the original matrices without reordering rose up to 2.1x, achieving a better behaviour as the number of threads increases. The average improvement was about 2.6% for the sequential case and 14.1% using 8 threads.

– **Another reordering development contributed in this thesis has consisted in using the latency information provided by the EARs to improve the behaviour of the data reordering technique.** Sets of consecutive rows were considered. Whereas the original technique takes into account only distances among the rows of the original sparse matrix, a new developed criterion uses the maximum sampled latency per row. According to this criterion, two consecutive rows $x$ and $x+1$ are included within the same set if the following condition is met: $Latency(x+1) < threshold$. Two different

thresholds have been studied: average and 7-cycle latency. Once the windows are defined, the original locality optimisation technique is applied obtaining the reordering matrix. We have demonstrated that rows with no sampled entries in the latency histogram are likely to present small size and low latency. Therefore, if a row with no sampled entries is included together with its previous row within the same set, this procedure is expected to be a good approximation. Results are very encouraging, obtaining noticeable improvements for several matrices, which proves that reordering guided by latencies can overcome the original technique in terms of performance. The advantages of this method comprise a low overhead and low variability in the sampling results.

– The adequate placement of data is essential to improve the performance of a program. One of the objectives of this work was the study of hardware counter-based optimisation techniques for page migration. The last contribution of this thesis has been **the development of a user-level, page migration software infrastructure based on hardware counters.** It supports **two migration strategies** that rely on the accurate information provided by the EARs. The first one is a competitive algorithm based exclusively on the number of accesses to a page from any cell. The second one refines the former using information about the access latency. The infrastructure performs a statistical profiling and migrates pages when needed according to the migration algorithm in use. Formally formulated for a $N - cell$ system, the algorithms have been tested on a FINISTERRAE rx7640 node ($N = 2$). The experimental results show that a sampled, EAR-based profile is enough to obtain a representative portrayal of data accesses on each cell. Our tests depicted two scenarios: a first one, in which a set of benchmarks is evaluated on a dedicated environment, and a second one in which a multiprogrammed environment is considered. The latter uses an interfering program to increase the need for data reallocation, since the OS scheduler migrates threads to remote cells in the presence of other programs.

  The results show a noticeable reduction of remote accesses and execution time, achieving speedups of up to 13% for the dedicated environment and 30% for the multiprogrammed one. Despite that peak improvement in the former, we verified that it is usually hard to beat the results of the system's first-touch allocation policy in a dedicated environment when considering efficient parallel codes. The improvements were scarce and irregular, and only those benchmarks with an important data movement got benefited from our migration algorithms. In the multiprogrammed scenario, however,

the improvement overwhelmed the overhead introduced by the migration infrastructure and the speedup was higher and more regular.

So far, the results achieved show a promising outlook, since some improvements have been obtained even in scenarios in which there were little room for them. After this work, several lines of research remain open and, as we continue the research presented in this dissertation, our efforts will focus on the following points:

– **Applicability of our locality improvement strategies to other architectures:** PEBS is a feature that has recently been included in most of the Intel Core-based processors. A similar solution exists in the AMD platform, called *Instruction-based sampling* (IBS) [86]. We intend to test our strategies in other NUMA architectures based on those processors. So far, we have carried out some research in an Intel Nehalem platform with satisfactory results. In the same line, we intend to study the PEBS capabilities of the new *Performance Counters for Linux* (PCL), the *de facto* standard monitoring framework embedded in the most recent Linux kernel versions. In these versions, `libpfm` has abandoned the use of `Perfmon` as the interface with the PMU and assists PCL instead to provide PEBS capabilities. This point can also be applied to the page migration techniques presented in this dissertation.

– **Study of the possibilities that the Roofline Model offers to assist data locality decisions in runtime:** The synergistic relationship between hardware counters and the Roofline Model suggests that a real-time, dynamic Roofline Model could be able to inspect and make suggestions about thread and data allocations to improve the performance of a running application.

– **Improvement of the page migration infrastructure and new scenarios:** The user-level approach of the page migration infrastructure provides the user with an ease-of-use, transparent application. However, this advantage may also become a weakness: its overhead is likely to be higher than a kernel-level application. Some of the improvements of the latency algorithm would have been better had it not been for some overhead introduced by the algorithms or ping-pong effects on the pages to migrate. These facts are worthy of study, since there is room for improvement and more efficient algorithms might achieve a higher performance.

  On the other hand, although our migration algorithms have been developed for a *N-cell* NUMA system, they have only been tested on a two-cell node. We strongly believe

that the potential of the latency-based algorithms presented will reside in their ability to reduce the execution time in a several-level latency systems, such as the Superdome node. Once overcome the bureaucratic and technical barriers to configure properly such a node, we intend to carry out a study in this system to evaluate and refine our algorithms.

Additionally, as a further research, the page migration infrastructure could be provided with the capability to inspect the performance and ratio of pages migrated and disable itself or modify the sampling period when the monitored program seems to behave properly, in order to avoid unnecessary overheads. Moreover, this capability could be bound to new migration algorithms. Among them, the proposed dynamic Roofline Model, which would suggest the improvements to perform in runtime.

A last line of work to take into consideration is the inclusion of page replication techniques, together with techniques to combine thread and page migrations in order to keep each thread close to their dataset. In this regard, some research is currently being undertaken in our group.

At the outset of this dissertation, five goals were proposed. The work presented here meets those goals. The hardware counters have proved to be an effective vehicle for enhancing the locality of several applications in runtime, and the results obtained are encouraging to continue the research in the proposed lines.

# Resumen

*Siguiendo el reglamento de los estudios de tercer ciclo de la Universidad de Santiago de Compostela, aprobado en la Junta de Gobierno el día 7 de abril de 2000 (DOG de 6 de marzo de 2001) y modificado por la Junta de Gobierno del 14 de noviembre de 2000, el Consejo de Gobierno del 22 de noviembre de 2003, del 18 de julio de 2005 (artículos 30 a 45), del 11 de noviembre de 2008 y del 14 de mayo de 2009; y, concretamente, cumpliendo las especificaciones indicadas en el capítulo 4, artículo 30, apartado 3 de dicho reglamento, se muestra a continuación un resumen en castellano de la tesis.*

En los últimos años hemos estado asistiendo a una evolución en los recursos computacionales para uso científico y de ingeniería. La Ley de Moore se ha mantenido a costa de integrar varios núcleos por procesador y buscar nuevas configuraciones en supercomputación. La línea que tradicionalmente ha marcado la diferencia entre multicomputadores, entendidos como nodos monoprocesador con memoria privada conectados en red, y multiprocesadores, o equipos con varios procesadores compartiendo memoria, es cada vez más difusa. Los nuevos supercomputadores se organizan en una disposición de *constelación*, en la cual cada nodo de un conjunto conectado por una red de alta velocidad es, a su vez, un multiprocesador de memoria compartida en el cual podemos encontrar varios procesadores multinúcleo.

Centrándonos en el caso de los sistemas multiprocesador, NUMA (*Non Uniform Memory Access*) es la disposición de memoria compartida más común hoy en día. La ventaja de proporcionar a todos los procesadores una visión uniforme del espacio de memoria tiene como inconveniente el mayor coste de acceso a aquellas direcciones alojadas en memorias físicamente remotas, no locales. En este contexto, la interacción de los sistemas de coherencia y consistencia caché, la jerarquía de memoria, y la influencia de buses y procesadores es compleja y está lejos de resultar intuitiva. Modelar el comportamiento de un código paralelo que corra en

este tipo de arquitecturas pasa por realizar un *profiling*[1] de la aplicación. Esta técnica permite averiguar dónde se encuentran los cuellos de botella de un programa. Es decir, cómo y dónde emplea la mayor parte de su tiempo de ejecución. De este modo, la información recabada permitirá tomar decisiones, por ejemplo, para la mejora de la localidad y el balanceo de datos de un programa, con el objetivo de reducir su tiempo de ejecución.

Un tipo de códigos computacionales en los que la localidad de datos es particularmente importante es el de los códigos irregulares. Un código irregular, en contraposición a uno regular, es aquel que presenta indirecciones en sus accesos a memoria de forma que impide averiguar, en tiempo de compilación, el conjunto de posiciones accedidas. Estos códigos presentan baja localidad y escaso reuso de la jerarquía de memoria. Un ejemplo típico de código irregular, abordado en esta tesis y presente en numerosas aplicaciones científicas y resolutores de sistemas de ecuaciones de métodos iterativos, es el producto matriz dispersa – vector (SpMV). Debido a la importancia de este kernel, es posible encontrar en la literatura diversas técnicas para la mejora de su localidad [4, 5, 6]. La creciente diferencia entre las velocidades de acceso a la memoria y de procesamiento hacen que la reordenación adecuada del patrón de accesos a datos de este tipo de kernels cobre cada vez mayor relevancia a la hora de mejorar su rendimiento en tiempo de ejecución.

Por otra parte, en la actualidad, los *contadores de monitorización de rendimiento*, también denominados *contadores hardware*, constituyen una importante herramienta de monitorización incluida en la PMU (unidad de monitorización de rendimiento) [3] de la mayoría de los microprocesadores modernos. Con una sobrecarga prácticamente imperceptible, permiten inspeccionar de forma no intrusiva un proceso en ejecución. En los últimos años han aparecido contadores de mayor precisión que permiten realizar tareas de PEBS (muestreado de precisión basado en eventos) [10], los cuales proporcionan, por ejemplo, las direcciones exactas del puntero de instrucción y de los datos accedidos en memoria para los que ha ocurrido un determinado evento.

En esta tesis se va un paso más allá de la utilización clásica de los contadores hardware como herramienta de monitorización, y se proponen nuevas técnicas para usarlos activamente en la toma de decisiones, de forma que conlleven una mejora de la localidad –y, por ende, del rendimiento– de diferentes tipos de aplicaciones. La plataforma en la que se enmarca este trabajo es el supercomputador FINISTERRAE, ubicado en el Centro de Supercomputación de Galicia (CESGA) [50]. Cada uno de sus nodos contiene 8 microprocesadores dual-core

[1] Se ha mantenido el término original en inglés por estar ampliamente aceptado y no existir un equivalente en castellano.

Itanium2 Montvale. Las PMU de cada uno de los núcleos de los Itanium2 Montvale integran un tipo particular de contadores hardware, los EAR (*Event Address Register*), que permiten realizar un PEBS y proporcionan información de las muestras obtenidas como, por ejemplo, la dirección accedida en memoria cuando se produce un evento determinado (ej: un fallo de la caché de segundo nivel), la latencia de dicho acceso, o la posición del puntero de instrucción.

En esta tesis se aborda el problema de mejora de la localidad en dos vertientes. Por un lado, se ha considerado el problema de la mejora de rendimiento de códigos irregulares, así como la reducción del coste asociado a técnicas de reordenación para éstos, en tiempo de ejecución. Por otro, se han desarrollado técnicas de migración dinámica de páginas para mejorar la localidad de códigos paralelos en tiempo de ejecución. En ambos casos, la información proporcionada por los contadores hardware y, en particular, los EARs, ha sido esencial para poder tomar decisiones de forma activa en tiempo de ejecución. Para la consecución de este trabajo se definieron los siguientes objetivos específicos:

– *Evaluación de la arquitectura*: En primer lugar es necesario obtener un conocimiento detallado de los nodos que componen el supercomputador FINISTERRAE, entendiendo cómo se comportan determinados programas en una arquitectura tan compleja.

– *Estudio de los contadores hardware*: A continuación, se debe profundizar el sistema de monitorización y muestreo de la PMU de los Itanium2 Montvale. Ello incluye familiarizarse con las librerías y herramientas disponibles para tal fin.

– *Mejora de técnicas de localidad*: Aprovechando las características de muestreo ofrecidas por los contadores hardware, se estudiará cómo reducir el coste de determinadas técnicas de mejora de la localidad para códigos irregulares.

– *Infraestructura de migración dinámica de páginas*: Con el fin de proponer estrategias de migración dinámica de páginas, es necesario previamente llevar a cabo el desarrollo de una infraestructura software capaz de monitorizar, muestrear y modificar la ejecución de un programa.

– *Definición de estrategias de migración de páginas*: Una vez que se disponga de la infraestructura de migración de páginas, se podrán proponer diferentes estrategias de migración basadas en la información proporcionada por los contadores hardware.

Los siguientes apartados detallan los objetivos y los resultados obtenidos.

## Evaluación de la arquitectura del FINISTERRAE

FINISTERRAE es uno de los supercomputadores de memoria compartida más grandes de Europa. Está compuesto por 142 nodos de computación HP-Integrity rx7640. Cada uno de estos nodos comprende dos celdas con 4 microprocesadores dual-core Itanium2 Montvale, conectados por parejas a un bus y a una memoria propia a través de un controlador de celda (ver Figura 2.2 en Página 15). La coherencia caché entre núcleos conectados a un mismo bus se mantiene mediante un protocolo de *snooping*, mientras que las coherencias entre buses y entre celdas son mantenidas por un directorio ubicado en memoria. La memoria total de cada nodo, en disposición NUMA, es de 128 GB (dos módulos de 64 GB). FINISTERRAE cuenta también con un nodo adicional formado por 128 núcleos y 1 TB de memoria.

Los Intel Itanium2 serie 9100 (Montvale) son procesadores RISC de doble núcleo. Cada uno de los núcleos disponen de jerarquía propia de caché en tres niveles desde la L1 a la L3. Los Itanium2 son procesadores que implementan un conjunto de instrucciones EPIC (*Explicitly Parallel Instruction Computing*), en la cual la responsabilidad de reordenar las instrucciones en el pipeline para maximizar la ejecución en paralelo recae en el compilador, al contrario que en los sistemas tradicionales en los que era el propio procesador quien debía asumir esa tarea en tiempo de ejecución.

Centrándonos en la unidad de monitorización, o PMU, de cada núcleo, podemos encontrar un nuevo tipo de contadores hardware denominados *Event Address Registers* (EAR). En esta tesis se ha utilizado la información proporcionada por los DEAR, o EAR de datos. Estos contadores son usados únicamente en modo de muestreo (en contraposición a los contadores tradicionales que, por contaje, dan un valor del número de eventos ocurridos) para obtener un conjunto de muestras de un programa que está ejecutándose en uno de los núcleos del procesador. Los EAR se pueden utilizar para realizar un muestreo basado en eventos (EBS). Es decir, cuando ocurre un número preprogramado de eventos, se captura una muestra. El tipo de eventos que pueden ser capturados son: fallos caché L1, cargas de datos de punto flotante, fallos de la TLB de primer nivel, y fallos de ALAT. La peculiaridad de estos contadores es que, a diferencia de los tradicionales, cada muestra capturada contiene información precisa sobre la instrucción ejecutada, dirección virtual del dato accedido, latencia y *timestamp* de dicho acceso.

Existen varios factores que pueden afectar al rendimiento de un programa paralelo ejecutado en el FINISTERRAE por tratarse de un sistema NUMA. Entre otros, la asignación de threads a núcleos que hace el planificador del sistema operativo, los mecanismos de coheren-

cia y consistencia caché, y la afinidad de los datos en memoria a los núcleos. El comportamiento de un nodo rx7640 de FINISTERRAE fue evaluado para códigos densos y dispersos. Se encontró que cada núcleo del nodo se comporta como si fuera un microprocesador independiente, en lo cual influye la jerarquía propia de caché de que dispone cada núcleo. Por otra parte, se constató la importancia de la ubicación de threads a núcleos. Nuestros experimentos mostraron que cada controlador de celda introduce cierto retardo, lo cual haría recomendable situar los threads de una aplicación en núcleos conectados al mismo bus. Sin embargo, la sobrecarga de compartir el bus supera fácilmente el retardo mencionado, por lo que se encontró que, cuando el tamaño de los datos a manejar por cada thread supera el tamaño de la caché L3, es más adecuado situar los threads en núcleos de buses diferentes y, si es posible y existe localidad de datos en memoria, en celdas diferentes. Esta situación fue corroborada por las decisiones del planificador de Linux que, aun sin presentar un comportamiento determinista, tiende a ubicar los threads en núcleos de distintas celdas.

Por otro lado, es frecuente que un programa paralelo comience con un thread maestro ubicando datos en la memoria de una celda y, a continuación, cree el resto de threads, que serán dispersados por todo el nodo. Linux cuenta con un sistema de ubicación de datos *first-touch*, por el cual ubica los datos permanentemente en la memoria del thread que primero accede a ellos. En estos casos, parte de los threads accederán remotamente a los datos. Sin disponer de mecanismos de migración de páginas, y mientras el número de threads lo permita, resulta adecuado mantener la ubicación de los threads en núcleos de la misma celda.

Adicionalmente al trabajo anterior, se implementó un modelo de rendimiento denominado *modelo Roofline* para el FINISTERRAE [60]. Existen varios modelos estadísticos que permiten predecir de forma precisa el rendimiento de un programa en un sistema. Sin embargo, normalmente son difíciles de utilizar por usuarios no expertos y raramente proporcionan información que permita mejorar el rendimiento del programa. El modelo Roofline proporciona, de manera gráfica, predicciones realistas del rendimiento y de la productividad de un sistema, informando de los cuellos de botella y sugiriendo posibles modificaciones para mejorar el rendimiento de la aplicación. Integra en una única gráfica el rendimiento de cada núcleo, el ancho de banda del sistema y la localidad de datos para un programa determinado. En esta tesis se implementó el modelo Roofline para un nodo rx7640 del FINISTERRAE y se testeó con el producto matriz dispersa-vector. Los resultados muestran cómo este código está limitado por el ancho de banda, no por la capacidad computacional del sistema, y sugiere qué modificaciones podrían realizarse para mejorar su rendimiento en cuanto a reubicación

de threads en núcleos y reordenación de datos, proporcionando además información acerca del máximo rendimiento que se puede esperar.

## Estudio de los contadores hardware

Para poder acceder a la información proporcionada por los EAR en los Itanium2 Montvale es necesario utilizar algún tipo de interfaz software que proporcione una capa de abstracción y facilite su programación. Tras realizar una evaluación de las opciones disponibles, se optó por usar la interfaz `Perfmon`. `Perfmon` se instala como un parche en el kernel de Linux. Su librería asociada `libpfm`, permite al programador acceder a todas las características de la PMU de cada núcleo del Itanium2 Montvale. Un programa puede ser automonitorizado, si contiene el código que le permite obtener información sobre su propia ejecución, o ser monitorizado por otro programa. `Perfmon` se ha utilizado habitualmente en modo de muestreo. Para ello, utiliza un *buffer* de almacenamiento de datos a nivel de kernel que, cuando se llena, genera una interrupción y se mapea a espacio de usuario para poder ser leído por el programador. De este modo, se reduce la sobrecarga asociada a las interrupciones. `Perfmon` fue evaluado en el FINISTERRAE con un producto matriz dispersa-vector paralelo y con un kernel denso, utilizando un conjunto de matrices de prueba como entrada. El evento capturado fue DATA_EAR_CACHE_LAT4, que muestrea aquellos fallos caché de L1 cuya latencia de resolución tarde más de 4 ciclos. Como en nuestros códigos trabajamos con valores en punto flotante, y la caché L1 es de enteros, todos los accesos a valores de punto flotante se consideran fallos de L1 y son susceptibles de ser capturados por `Perfmon`, dependiendo del período de muestreo empleado. Los resultados muestran que ni el patrón de acceso ni el número de threads influyen en el porcentaje de valores muestreados. Por otro lado, se estimó la sobrecarga introducida en un código automonitorizado por `Perfmon`, concluyendo que, en promedio, la sobrecarga es de un 0.01%, perfectamente asumible para nuestro trabajo.

## Mejora de técnicas de localidad

Una vez evaluado el entorno de trabajo y las herramientas de monitorización, se pasó a estudiar cómo utilizar los contadores hardware en técnicas de mejora de la localidad de códigos irregulares en sistemas paralelos. Un caso particular de estas técnicas son aquellas que modifican la ubicación de las estructuras de datos en memoria. Por lo general, el coste de estas técnicas depende del tamaño de los datos de entrada y no es hasta que se han ejecutado

varias iteraciones del programa que se compensa el coste inicial de la etapa de reordenación de los datos. Para nuestro estudio se ha utilizado una técnica de reordenación previamente desarrollada en nuestro grupo de investigación. Esta técnica utiliza métodos heurísticos para reordenar las estructuras de datos de un código irregular, por ejemplo, de una matriz dispersa. Como ejemplo de código irregular se utilizó el producto matriz-vector, por tratarse de un código paradigmático de los irregulares que presenta bajo reuso de datos causado por el patrón de accesos con indirecciones.

En primer lugar, se estudió la capacidad de la técnica de reordenación de estimar la localidad de una matriz dada a partir de un conjunto incompleto de datos (es decir, muestreando aleatoriamente de forma manual la matriz de entrada), con el objeto de determinar la cantidad de información requerida para obtener resultados similares a la técnica original. Se realizaron estudios con 1%, 2%, 5%, 10%, 15% y 20% de la cantidad de elementos no nulos de las matrices originales de un conjunto de prueba. Esta información se utilizó para generar un vector de permutaciones para reordenar las filas en cada matriz. Se comprobó que valores de un 1% del total de elementos de la matriz eran suficientes para conseguir una reordenación que obtenía resultados similares (con una diferencia máxima del 5%) a los de la técnica original, con la ventaja de una importante reducción en la sobrecarga del proceso de generación del vector de permutaciones.

El mismo proceso se repitió utilizando información parcial obtenida de un muestreo con los EAR de los Itanium2 Montvale. Dado que, con la configuración utilizada, los contadores obtienen únicamente la dirección de una posición de memoria accedida en el SpMV, no es posible, a priori, obtener la posición de la matriz que estaba siendo accedida cuando se produjo ese evento. Por ello, se desarrolló una técnica que hemos denominado DAST (*Dual Array Sampling Technique*), la cual permite obtener con precisión la fila y columna del elemento de la matriz muestreado. Tras generar los vectores de permutación con esta información muestreada para el conjunto de matrices de prueba, la técnica de reordenación obtuvo resultados similares a la técnica original pero, de nuevo, con una sobrecarga menor debido al uso de información muestreada. En general, con las técnicas utilizadas se consigue una reducción en el tiempo necesario para reordenar de entre el 93 y el 98% con respecto a la técnica original. Por otro lado, se han observado menores fluctuaciones en los resultados utilizando muestras procedentes de los contadores hardware frente a los valores muestreados aleatoriamente de forma manual.

En la misma línea, se desarrolló otro método para mejorar el rendimiento de la técnica de reordenación. Utilizando DAST, y aprovechando la información de la latencia de cada

acceso proporcionada por los contadores hardware, se modificó la heurística de decisión para reordenar la matriz original. En lugar de utilizar la distancia entre filas y/o columnas para decidir cómo agruparlas para reordenar la matriz, se usaron los valores máximos de latencia a cada fila. Se asume que, en el SpMV, si aparecen accesos al vector $X$ con latencia alta, quiere decir que hay poco reuso de esos elementos accedidos por los elementos no nulos de una fila y de las filas precedentes de la matriz. Por tanto, varias filas con latencias de acceso por debajo de determinado umbral se asignan conjuntamente para conseguir la reordenación de la matriz más eficientemente. Con esta técnica, se consiguieron mejoras en el rendimiento del SpMV de hasta un 30%.

## Técnicas de migración de páginas

Otro de los campos abordados en esta tesis para la mejora de la localidad ha sido el de la migración dinámica de páginas. La información de latencia asociada a las muestras obtenidas por los EAR permite obtener una idea precisa del coste y número de accesos realizados a una página de memoria durante la ejecución de un programa.

En primer lugar, se desarrolló una infraestructura software de monitorización y migración de páginas a nivel de usuario para aplicaciones paralelas. Su funcionamiento es el siguiente: un *programa monitor* recibe como parámetro el programa a monitorizar. Tras lanzarlo, detecta la creación de cada hilo y, mediante `Perfmon`, asigna un *contexto* a cada uno. Los contextos son estructuras lógicas de `Perfmon` que almacenan el estado de la PMU y otros eventos relacionados del sistema. Cada contexto es asociado a la PMU del núcleo en el que está ubicado cada thread del programa monitorizado. Si éste es migrado a otro núcleo durante su ejecución, el contexto lo sigue. El programa monitor configura las PMUs de cada núcleo para que muestreen un evento determinado proporcionado por los EARs. En este caso, el evento seleccionado es `DATA_EAR_CACHE_LAT4` que, como se explicó previamente, captura aquellos fallos caché de L1 cuya latencia de resolución tarde más de 4 ciclos. `Perfmon` crea y asocia un buffer a nivel de kernel a cada uno de los contextos, que almacenan los datos muestreados (ver Figura 5.2 en página 108). Una vez configuradas las PMUs, el programa a monitorizar es iniciado. Durante su ejecución, cada vez que se produce un desbordamiento del contador hardware asociado a un EAR, el evento que lo provocó y su información relacionada son almacenados como una entrada en el buffer. Cuando uno de los buffers se llena, envía una notificación al programa monitor, el cual lee y procesa la información recogida (ver Figura 3.3 en página 59). Este proceso continúa durante un *período de monitorización*, al final del cual el

programa monitor utiliza la información recabada para, basándose en las decisiones tomadas por una estrategia de migración dada, mover determinadas páginas de una memoria a otra mediante la función `move_pages()` del sistema operativo. Una vez realizada la migración, el período de monitorización se inicia de nuevo. Nótese que una de las ventajas de esta infraestructura es que no requiere tener que modificar en absoluto las fuentes o los binarios de la aplicación monitorizada.

Las estrategias de migración propuestas se basan en modelos teóricos que desarrollamos para sistemas jerárquicos con varios niveles de latencia de acceso a memoria, como el nodo Superdome del FINISTERRAE. Estos modelos han sido particularizados y probados en los nodos rx7640, de dos celdas, utilizando los benchmarks NAS paralelos en OpenMP. El primer modelo de migración es un algoritmo competitivo, en el que la celda a la que se migra una página determinada es aquella cuyos procesadores acceden más veces en un período de monitorización dado. Esta migración se justifica asumiendo que el número de accesos desde cada celda se mantendrá constante, al menos, en el siguiente período de monitorización. El segundo modelo de migración se basa principalmente en las latencias de acceso en lugar de en el número de accesos, considerando la carga en tiempo real de cada celda y moviendo las páginas a aquella cuya latencia media estimada se prevé menor. Para ello, es necesario determinar dicha previsión. Estas estrategias han sido probadas en un entorno dedicado con 4, 8, 12 y 15 threads, en el que la única aplicación ejecutándose era uno de los benchmarks. En este tipo de entornos es difícil batir a la política *first-touch* del planificador del sistema operativo sobre códigos ya eficientes de por sí como los NAS, ya que una vez que cada thread es asignado a un núcleo y se ubican sus datos en la memoria local, generalmente se producen pocas migraciones de threads que fuercen accesos remotos a memoria. Aún así, en aquellos benchmarks con movimiento importante de datos, se consiguieron ciertas mejoras. En un segundo experimento, se probaron las estrategias de migración en un entorno multiprogramado, lo que supone una situación más realista en la que varios programas deben compartir los recursos hardware disponibles. En dicho experimento, otro programa paralelo se ejecutaba a intervalos en una u otra celda mientras se ejecutaba el monitorizado, forzando migraciones en los threads del benchmark monitorizado que fueron compensadas con el mecanismo de migración de páginas. En este caso, las mejoras fueron más importantes, consiguiendo valores de speedup en algunos casos de hasta el 30%.

# References


[1] N. Sánchez, "HP Integrity rx7640 and rx8640 server. Technical presentation," *Hewlett-Packard*, 2006.

[2] Hewlett Packard, *Dual-Core Update to the Intel Itanium 2 Processor Reference Manual*, 2006. Technical paper.

[3] D. Mosberger and S. Eranian, *IA-64 Linux Kernel: Design and Implementation*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[4] L. Rauchwerger, N. M. Amato, and D. A. Padua, "Run-time methods for parallelizing partially parallel loops," in *Proceedings of the 9th ACM International Conference on Supercomputing*, LNCS, pp. 137–146, ACM press, 1995.

[5] R. Eigenmann, J. Hoeflinger, and D. Padua, "On the automatic parallelization of the perfect benchmarks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 1, pp. 5–23, 1998.

[6] E. Gutiérrez, O. Plata, and E. L. Zapata, "A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors," in *Proc. of the Int. Conf. on Supercomputing*, LNCS, pp. 78–87, ACM SIGARCH, Springer-Verlag, May 2000.

[7] E. Gutiérrez, O. G. Plata, and E. L. Zapata, "An analytical model of locality-based parallel irregular reductions," *Parallel Computing*, vol. 34, no. 3, pp. 133–157, 2008.

[8] E. Herruzo, G. Bandera, O. G. Plata, and E. L. Zapata, "Reducing cache misses by loop reordering," in *Proceedings of the International Conference ParCo*, pp. 541–548, 2005.

[9] R. Das, M. Uysal, J. Saltz, and S. Y. Hwang, "Communication optimizations for irregular scientific computations on distributed memory architectures," *Parallel and Distributed Computing*, vol. 22, no. 3, pp. 462–478, 1994.

[10] Precise Event-Based Sampling (PEBS). http://perfmon2.sourceforge.net/pfmon_intel_core.html#pebs.

[11] E. Cuthill and J. McKee, *Several strategies for reducing the bandwidth of matrices*. Rose and Willoughby, 1972.

[12] J. C. Pichel, D. E. Singh, and J. Carretero, "Reordering algorithms for increasing locality on multicore processors," in *Proc. of the IEEE Int. Conf. on High Performance Computing and Communications*, pp. 123–130, 2008.

[13] P. R. Amestoy, T. A. Davis, and I. S. Duff, "An approximate minimum degree ordering algorithm," *SIAM Journal on Matrix Analysis and Applications*, vol. 17, no. 4, pp. 886–905, 1996.

[14] L. Oliker, X. Li, P. Husbands, and R. Biswas, "Effects of ordering strategies and programming paradigms on sparse matrix computations," *SIAM Review*, vol. 44, no. 3, pp. 373–393, 2002.

[15] A. L. G. A. Coutinho, M. A. D. Martins, R. M. Sydenstricker, and R. N. Elias, "Performance comparison of data-reordering algorithms for sparse matrix–vector multiplication in edge-based unstructured grid computations," *International Journal for Numerical Methods in Engineering*, vol. 66, no. 3, pp. 431–460, 2006.

[16] B. B. Fraguela, R. Doallo, and E. L. Zapata, "Memory hierarchy performance prediction for blocked sparse algorithms," *Parallel Processing Letters*, vol. 9, pp. 347–360, Sept. 1999.

[17] J. J. Navarro, E. García, J. L. Larriba-Pey, and T. Juan, "Block algorithms for sparse matrix computations on high performance workstations.," in *Proc. IEEE Int'l. Conf. on Supercomputing (ICS'96)*, pp. 301–309, 1996.

[18] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck, "Sparse tiling for stationary iterative methods," *Int. J. High Perform. Comput. Appl.*, vol. 18, pp. 95–113, February 2004.

[19] E. J. Im, K. A. Yelick, and R. Vuduc, "SPARSITY: Framework for optimizing sparse matrix-vector multiply," *International Journal of High Performance Computing Applications*, vol. 18, pp. 135–158, February 2004.

[20] A. Buttari, V. Eijkhout, J. Langou, and S. Filippone, "Performance optimization and modeling of blocked sparse kernels," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 467–484, November 2007.

[21] R. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *High Performance Computing and Communications* (L. Yang, O. Rana, B. Di Martino, and J. Dongarra, eds.), vol. 3726 of *Lecture Notes in Computer Science*, pp. 807–816, Springer Berlin / Heidelberg, 2005.

[22] V. Karakasis, G. Goumas, and N. Koziris, "A comparative study of blocking storage methods for sparse matrices on multicore architectures," in *Computational Science and Engineering, 2009. CSE '09. International Conference on*, vol. 1, pp. 247–256, August 2009.

[23] M. Belgin, G. Back, and C. J. Ribbens, "Pattern-based sparse matrix representation for memory-efficient smvm kernels.," in *ICS'09*, pp. 100–109, 2009.

[24] S. Toledo, "Improving memory–system performance of sparse matrix–vector multiplication," in *Proc. of the 8th SIAM Conf. on parallel processing for scientific computing*, March 1997.

[25] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '99, (New York, NY, USA), ACM, 1999.

[26] E. J. Im and K. Yelick, "Optimizing sparse matrix vector multiplication on SMPs," in *Proc. of the 10th SIAM Conf. on parallel processing for scientific computing*, March 1999.

[27] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera, "Performance optimization of irregular codes based on the combination of reordering and blocking techniques," *Parallel Computing*, vol. 31, no. 8–9, pp. 858–876, 2005.

[28] K. Kourtis, G. I. Goumas, and N. Koziris, “Optimizing sparse matrix-vector multiplication using index and value compression.,” in *Conf. Computing Frontiers’08*, pp. 87–96, 2008.

[29] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, “Optimization of sparse matrix-vector multiply on emerging multicore platforms,” in *Proc. of Supercomputing (SC)*, 2007.

[30] S. Eranian, “What can performance counters do for memory subsystem analysis?,” in *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’08)*, MSPC ’08, (New York, NY, USA), pp. 26–30, ACM, 2008.

[31] J. Marathe, F. Mueller, and B. R. de Supinski, “Analysis of cache-coherence bottlenecks with hybrid hardware/software techniques,” *ACM Trans. Archit. Code Optim.*, vol. 3, pp. 390–423, December 2006.

[32] B. R. Buck and J. K. Hollingsworth, “Data centric cache measurement on the Intel Itanium 2 Processor,” *SC Conference*, vol. 0, p. 58, 2004.

[33] Y. Choi, A. Knies, G. Vedaraman, and J. Williamson, “Design and experience: Using the Intel Itanium2 processor performance monitoring unit to implement feedback optimizations,” in *Proc. of EPIC2 Workshop*, 2002.

[34] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O’Boyle, and O. Temam, “Rapidly selecting good compiler optimizations using performance counters,” in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO ’07, (Washington, DC, USA), pp. 185–197, IEEE Computer Society, 2007.

[35] M. M. Tikir and J. K. Hollingsworth, “Hardware monitors for dynamic page migration,” *Journal of Parallel and Distributed Computing*, vol. 68, pp. 1186–1200, September 2008.

[36] J. Antony, P. P. Janes, and A. P. Rendell, “Exploring thread and memory placement on numa architectures: Solaris and linux, ultrasparc/fireplane and opteron/hypertransport.,” in *HiPC’06*, pp. 338–352, 2006.

[37] M. M. Tikir and J. K. Hollingsworth, "Using hardware counters to automatically improve memory performance," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, (Washington, DC, USA), pp. 46–, IEEE Computer Society, 2004.

[38] J. Marathe and F. Mueller, "Hardware profile-guided automatic page placement for cc-NUMA systems," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, (New York, NY, USA), pp. 90–99, ACM, 2006.

[39] V. Thakkar, "Dynamic Page Migration on ccNUMA Platforms Guided by Hardware Tracing," Master's thesis, Graduate Faculty of North Carolina State University, 2008.

[40] J. M. Bull and C. Johnson, "Data distribution, migration and replication on a ccNUMA architecture," in *Proceedings of the Fourth European Workshop on OpenMP*, 2002.

[41] J. Tao, M. Schulz, and W. Karl, "Improving data locality using dynamic page migration based on memory access histograms," in *Proceedings of the International Conference on Computational Science-Part II*, ICCS '02, (London, UK, UK), pp. 933–942, Springer-Verlag, 2002.

[42] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé, "A case for user-level dynamic page migration," in *Proceedings of the 14th international conference on Supercomputing*, ICS '00, (New York, NY, USA), pp. 119–130, ACM, 2000.

[43] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé, "User-level dynamic page migration for multiprogrammed shared-memory multiprocessors," in *Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, ICPP '00, (Washington, DC, USA), pp. 95–, IEEE Computer Society, 2000.

[44] D. S. Nikolopoulos, C. D. Polychronopoulos, T. S. Papatheodorou, J. Labarta, and E. Ayguadé, "Scheduler-activated dynamic page migration for multiprogrammed DSM multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 62, no. 6, pp. 1069–1103, 2002.

[45] K. M. Wilson and B. B. Aglietti, "Dynamic page placement to improve locality in CC-NUMA multiprocessors for TPC-C," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, (New York, NY, USA), pp. 98–107, ACM, 2001.

[46] B. Goglin and N. Furmento, "Enabling high-performance memory migration for multithreaded applications on linux," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, (Washington, DC, USA), pp. 1–9, IEEE Computer Society, 2009.

[47] B. Goglin and N. Furmento, "Memory Migration on Next-Touch," in *Linux Symposium*, (Montreal, Canada), 2009.

[48] The Harwell-Boeing Sparse Matrix Collection. http://math.nist.gov/MatrixMarket/collections/hb.html.

[49] T. A. Davis, "The university of florida sparse matrix collection," *NA Digest, 92 (1994), NA Digest, 96 (1996), and NA Digest, 97 (1997).*

[50] Galicia Supercomputing Centre. http://www.cesga.es.

[51] J. Hennessy and D. Patterson, *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2003.

[52] Hewlett Packard technical white paper, *Inside the Intel Itanium 2 processor*, 2002.

[53] M. Burrell, "Writing Efficient Itanium 2 Assembly Code." https://wizardlike.ca/files/itanium.pdf, 2010.

[54] L. Marek, "Parallel processing and software performance," Master's thesis, Dept. of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, 2008.

[55] The numactl command. http://linux.die.net/man/8/numactl.

[56] IA-32 Intel Architecture Software Developers Manual. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[57] HP Integrity rx7640 Server Quick Specs. http://h18000.www1.hp.com/products/quickspecs/12470_div/12470_div.pdf.

[58] Y. Saad, *Iterative Methods for Sparse Linear Systems, 2nd edition*. Philadelpha, PA: SIAM, 2003.

[59] A. Kleen, "An numa api for linux," *SUSE Labs*, 2004.

[60] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, pp. 65–76, April 2009.

[61] L. McVoy and C. Staelin, "LMbench: portable tools for performance analysis," in *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 23–23, USENIX Association, 1996.

[62] Performance Application Programming Interface (PAPI). http://icl.cs.utk.edu/papi/.

[63] Pfmon performance monitoring tool. http://perfmon2.sourceforge.net/pfmon_users-guide.html.

[64] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "PIN: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, (New York, NY, USA), pp. 190–200, ACM, 2005.

[65] Paradyn Project. ParseAPI: An application program interface for binary parsing. http://paradyn.org/html/parse0.9-features.html.

[66] J. Levon et al., "OProfile." http://oprofile.sourceforge.net/.

[67] S. Eranian, *The perfmon2 Interface Specification. Technical Report HPL-2004-200R1*. HP Labs, February 2005.

[68] M. Pettersson, "The Perfctr interface." http://user.it.uu.se/mikpe/linux/perfctr.

[69] J. Reinders, *VTune Performance Analyzer Essentials*. Intel Press, 2005.

[70] Hewlett Packard, *HPCPI and Xtools Version 0.6.6 User's Guide*, 2008.

[71] R. Hundt, "Hp caliper: A framework for performance analysis tools," *IEEE Concurrency*, vol. 8, pp. 64–71, 2000.

[72] Performance Counters for Linux. https://lkml.org/lkml/2008/12/4/401.

[73] Perfcounters added to the mainline. http://lwn.net/Articles/339361.

[74] S. Eranian, "Perfmon2: a flexible performance monitoring interface for linux," in *Ottawa Linux Symposium (OLS)*, 2006.

[75] S. Eranian, D. Mosberger, J. Callister, and S. Fernando, "Performance profiling for fun and profit," in *Gelato Federation Meeting*, 24 May 2005.

[76] J. C. Pichel, *Técnicas de optimización de la localidad para códigos irregulares sobre arquitecturas multiprocesador y multithreading*. Phd thesis, September 2006.

[77] G. Gutin, A. Punnen, A. Barvinok, E. K. Gimadi, and A. I. Serdyukov, "The traveling salesman problem and its variations," 2002.

[78] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, "Finding Tours in the TSP," in *Institute for Discrete Mathematics, Universitat Bonn*, 1999.

[79] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, A. J. García-Loureiro, and F. F. Rivera, "Increasing the locality of iterative methods and its application to the simulation of semiconductor devices," *International Journal of High Performance Computing Applications*, vol. 24, no. 2, pp. 136–153, 2010.

[80] M. Galassi et al, *GNU Scientific Library Reference Manual (3rd Ed.)*. 2009.

[81] G. Karypis and V. Kumar, *METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices*. Univ. of Minnesota, Dept. of Computer Science/Army HPC Research Center, 1997.

[82] Perfmon2 monitoring interface and Pfmon monitoring tool. http://perfmon2.sourceforge.net.

[83] NAS Parallel Benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html.

[84] H. Jin, H. Jin, M. Frumkin, M. Frumkin, J. Yan, and J. Yan, "The OpenMP Implementation of NAS Parallel Benchmarks and its Performance," tech. rep., 1999.

[85] move_pages manual. http://linux.die.net/man/2/move_pages.

[86] P. J. Drongowski, “Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors (technical paper),” *Advanced Micro Devices, Inc.*, 2007.