Scalable System Software for High Performance Large-scale Applications

Alessandro Morari
Department of Computer Architecture
Universitat Politècnica de Catalunya

A thesis submitted for the degree of
Doctor of Philosophy

27 May 2014

Ai miei genitori Fabrizio ed Elena, a mia sorella Elisa ed ai miei nonni Stefano e Vera.

Acknowledgements

I would like to thank my Thesis Director Prof. Mateo Valero, for his continuous support and prompt attention, even when we were separated by a whole continent. I would also like to acknowledge my mentor Dr. Roberto Gioiosa, for believing in me from the beginning, and for challenging me with his thoughtful remarks. I would like to thank my friends and colleagues at Pacific Northwest National Laboratories, Dr. Oreste Villa and Dr. Antonino Tumeo. Their friendship and teachings made me change radically, transforming this time in the laboratory to an amazing experience. I would like to thank the director of the Center for Adaptive Supercomputer Software Dr. John Feo, for his incredible heart and honesty, and for supporting me from the first day we met. I would also like to thank my group leader at the Barcelona Supercomputing Center, Dr. Francisco Cazorla, for continuously supporting my research activity and for patiently dealing with my laziness with bureaucratic procedures. I would like to acknowledge my colleagues in the CAOS group at Barcelona Supercomputing Center Marco Paolieri, Roberta Piscitelli, Carlos Luque, Carlos Boneti, Bojan Maric, Kamil Kedzierski, Victor Jimenez, Petar Radojkovic, Vladimir Kacarevic, Dr. Eduardo Quiñones, Dr. Jaume Abella and Dr. Miquel Moreto for the interesting discussions and the beautiful moments shared when I was working in Barcelona. Also, I would like to thank my colleagues at IBM T. J. Watson Dr. Robert Wisniewski, Dr. Bryan Rosenburg, Dr. Jose Brunheroto, Dr. Chen-Yong Cher and Dr. Alper Buyuktosunoglu for sharing with me their knowledge and giving me the opportunity of working on extremely interesting problems. Finally, I am deeply grateful to my family for their incredible support, for their love and for giving me the gift of curiosity that has taken me where I stand today.

Abstract

In the last decades, high-performance large-scale systems have been a fundamental tool for scientific discovery and engineering advances. The next generation of supercomputers will be able to deliver a performance in the order of 10^{18} floating point operations per seconds (ExaFlop). The design of a new generation of supercomputers, called *Exascale* generation, is the object of a large part of current research in HPC. The availability of large-scale models and simulation and the availability of large instruments like telescopes, colliders and light sources has generated a growing volume of scientific data. Computing resources to move, analyze and manage this exponentially growing volume of data is becoming the next big challenge, commonly referred as *big data*. The steps toward the design of an Exascale Supercomputer are increasingly pointing to the necessity of integrating Exascale with big data.

Over the years, the difficulty of efficiently manage large-scale system resources has repeatedly shown the importance of providing scalable system software. System software has evolved to keep pace with technology and applications evolution. Research on scalable system software for large-scale system has historically been driven by two fundamental aspects: on one side, performance, efficiency and reliability, on the other side the goal of providing a productive developing environment.

Considered the challenges posed by the next-generation large-scale high-performance systems and applications, it is clear that system software needs to be significantly updated if not redesigned. In this Thesis, we propose approaches to measure, design and implement scalable system software.

Overall, this Thesis proposes three orthogonal approaches that should be integrated into a new system-software layer for next-generation large-scale high performance computing systems:

- *To obtain scalable applications a detailed measurement and under-*

standing of system software performance and overhead are necessary. We design and implement a methodology to provide detailed measurement of system software interruptions and their effect on applications performance.

- *The runtime system is the right candidate to provide highly scalable system services and to exploits low-level hardware optimizations.* Some system services could be moved from the operating system to the runtime system. A specialized runtime system for a class of application can leverage a deeper knowledge of the application behaviour and exploit low-level hardware features to improve scalability. We design and implement a runtime system for the class of irregular applications and show a performance improvement of several orders of magnitude with respect to more traditional approaches.
- *Exploiting low-level hardware features at user-level (i.e., in the application) can lead to considerable performance improvement.* We show two instances of optimizations leveraging architecture-specific hardware features and obtaining significant performance improvement.

To enhance the performance of an entire class of applications, those approaches should be integrated into a specialized runtime system.

Contents

List of Figures

Part I

Introduction and Backround

Chapter 1

Introduction

In the last decades, high-peformance large-scale systems have been a fundamental tool for scientific discovery and engineering advances. The sustained growth of supercomputing performance and the concurrent reduction in cost have made this techonology available for a large number of scientists and engineers working on many different problems.

1.1 Challenges of Next-generation High-performance Large-scale systems

Supercomputing performance has been growing at a faster pace than Moore's law, thanks to the ability to leverage systems scalability [10]. High Performance Computing (HPC) employs large clusters of multicore nodes, interconnected with a high-speed network. Today, scientific research and high-end technological firms leverage HPC computational resources to perform extremely complex computations in order to solve a large variety of problems. HPC is a necessary tool to face important scientific and technological challenges that might have a huge impact on human life. Some of the traditional HPC applications include: molecular dynamic, quantum chemistry, fluid dynamic, nuclear simulations. Besides traditional ones new applications are emerging, such as: improving therapy efficiency, extreme weather forecast, energy efficient aircraft design and material science analysis [25].

Furthermore, an entirely new paradigm of scientific discovery is emerging . The availability of large-scale models and simulations and the availability of very large instruments like colliders and light sources has generated a growing volume of scientific data [48]. This very large volume of data has the potential of further improving scientific research and our understanding of many problems. Nonetheless, large-scale computing systems were not designed to handle such a high data volume and are currently not efficient in this task. The design of next-generation supercomputers will include traditional HPC requirements as well as the new requirements to handle data-intensive computations.

Data intensive applications will hence play an important role in a variety of fields, and are the current focus of several research trends in HPC. A particular class of data-intensive applications is the class of irregular applications [67]. Applications of this class are called irregular because they may have irregular memory access pattern, irregular network communication and irregular instruction control logic. The unpredictable network and data access pattern has a significant impact on performance. In fact, modern large-scale HPC machine are not optimized for this type of applications and their performance results sub-optimal [159]. A large number of relevant applications fall in this class: bioinformatics, big science applications, complex network analysis, community detection, data analytics, natural language processing, pattern recognition, semantic databases and, in general, knowledge discovery applications. The challenges to support this class of applications is affecting the design of next-generation HPC hardware and software. Large projects [153] have been following this approach. Current large-scale systems focus around regular computations and data access patterns and exploit complex cache-based architectures to reduce data access latencies.

The solution to this kind of problems includes a complete redesign of the whole software stack. Being at the bottom of the software stack, the system software is expected to change drastically to support the upcoming hardware and to meet new application requirements [102].

Regarding the operating system (OS), two different approaches have been taken: on one side adapting a general purpose operating system (e.g. Linux), on the other side developing a specialized OS for HPC. The former solution leverages the large software ecosystem and experience available for general purpose OS (GPOS), while the latter solution guarantees better performance.

One of the drawbacks of using a GPOS is the difficulty of obtaining predictable performance [140]. Linux, for instance, is not designed to provide a constant predictable execution time to the applications. It features a variety of system services that can interrupt the application at any moment. This phenomena, called OS noise [137], has been frequently identified as an issue for large-scale systems. OS noise can, indeed, significantly reduce applications scalability. With a GPOS a detailed understanding of OS noise and its effects on applications performance is necessary. Once measured, OS noise can be reduced by re-configuring or modifying the OS.

Operating systems that are specialized for HPC do not show significant OS noise, because they are designed to be lightweight (so called lightweight kernels). Lightweight kernels (e.g. IBM CNK) guarantee predictable performance and optimal scalability [148]. The drawback of lightweight kernels is the lack of many features offered by a GPOS (e.g. full featured dynamic memory allocation).

Because of the need for a low-overhead and low-noise OS, many features traditionally provided by the OS are being moved into the runtime system. The runtime system, because of the higher knowledge of the running application, is the best candidate to provide services that depend on the application specific characteristics.

1.2 Proposed approach and methodologies

Considered the challenges posed by the next-generation large-scale high-performance systems and applications, it is clear that system software needs to be significantly updated, if not redesigned. In this Thesis, we propose an approach to measure, design and implement scalable system software. The ideas behind this work are the following:

- **Detailed measurement of system software overhead and its effect on scalability is necessary:** obtaining scalable system software can be achieved only throughout a detailed measurement and understanding of system software performance, overhead, and their effect on real applications. Without a deep understanding of the sources of OS noise scalable system software is difficult to design [89].

- **Adaptive system software:** system software has to adapt to the changing hardware architecture and applications. The use of a standard, full-featured OS it is not a viable solution anymore. Similarly, the exclusive use of a minimal lightweight kernel is not the optimal solution to support the wide range of next-generation applications [135]. These two approaches have to be merged in a modular adaptive solution that can provide the benefits of both.

- **The runtime system is the right candidate to provide highly scalable system services and to exploit low-level hardware optimizations:** some of the system services traditionally provided by the OS have to be moved into the runtime system. As shown in Figure 1.1 next-generation runtime systems will include more system services and will exploit low level hardware features to optimize performance and power [13].

Services like scheduling, power management and data locality management can be effectively provided by the runtime system, with the advantage of specializing them for a particular class of applications (e.g. irregular applications). Moreover, the runtime system is the software layer that can better exploit hardware features provided by modern microprocessors such as on-chip communications and hardware thread priorities. This approach allows the implementation of highly adaptive system software to face the scalability and power efficiency challenges of the next-generation large-scale machines [58]. Exploiting low-level hardware features such as fine-grain cache locality management, core-to-core communications and hardware thread priorities can significantly improve application performance. Nonetheless, optimizing a single application can require considerable effort. The runtime system, on the other hand, has detailed knowledge of the application behavior and requirements. For this reason, the runtime system is one of the best options to implement the optimization logic that exploit low-level hardware features to optimize an entire class of applications.

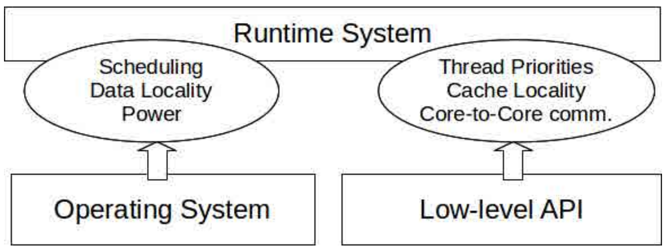


Figure 1.1: Next-generation runtime systems will include additional system services and exploit low-level hardware features to optimize an entire class of applications.

1.3 Thesis contributions

This Thesis address the problem of evaluating and designing scalable system software for large-scale system. This work starts by addressing the scalability issues of the OS, first regarding GPOS and then regarding lightweight kernels. Subsequently, we focus on the runtime system. For shared memory systems we show how a runtime system can be effectively used to improve data locality on emerging many-core architecture. Finally, we implement a runtime system for distributed memory systems that includes many of the system services required by next-generation applications.
The contribution of this thesis are as follows:

1. **Operating System Scalability**
   We provide an accurate study of the scalability problems of modern Operating Systems for HPC.

   OS noise measurement and tracing: We design and implement a methodology whereby detailed quantitative information may be obtained for each OS noise event. We validate our approach by comparing it to other well-known standard techniques to analyze OS noise, such FTQ (Fixed Time Quantum [154]). We provide a case study in which we use our methodology to analyze the OS noise when running benchmarks from the Lawrence Livermore National Lab (LLNL) Sequoia applications.

   Evaluation of the address translation management: we provide an apples-to-apples comparison of different TLB management approaches — dynamic memory mapping, static memory mapping with replaceable TLB entries, and static memory mapping with fixed TLB entries (no TLB misses) — on a real BG/P system with up to 1024 nodes (4096 cores).

2. **Runtime System Scalability**
   We show that a runtime system can efficiently incorporate system services and improve scalability for a specific class of applications.

   Design and implementation of a runtime library to efficiently execute irregular applications on a commodity cluster: we design and implement a full-featured runtime system and programming model to execute irregular applications on a commodity cluster. The runtime library is called Global Memory and Threading library (GMT) and integrates a locality-aware Partitioned Global Address Space communication model with a fork/join program structure. It supports massive lightweight multithreading, overlapping of communication and computation and small messages aggregation to tolerate network latencies. We compare GMT to other PGAS models, hand-optimized MPI code and custom architectures (Cray XMT) on a set of large scale irregular applications: breadth first search, random walk and concurrent hash map access. Our runtime system shows performance orders of magnitude higher than other solutions on commodity clusters and competitive with custom architectures.

3. **User-level Scalability Exploiting Hardware Features**
   We show the high complexity of low-level hardware optimizations for single applications, as a motivation to incorporate this logic into an adaptive runtime

system.

Exploiting hardware multi-thread priorities in multithreaded architectures: we evaluate the effects of controllable hardware-thread prioritization mechanism that controls the rate at which each hardware-thread decodes instruction on IBM POWER5 and POWER6 processors.

Exploting cache-locality and network-on-chip to optimize sorting in many-core architectures: we show how to effectively exploit cache locality and network-on-chip on the Tilera many-core architecture to improve intra-core scalability.

1.4 Thesis structure

The thesis is divided into four parts:

Part I includes the Introduction and Chapter 2 that provides the background for this work.

Part II addresses the scalability problems of modern Operating Systems for HPC. Chapter 3 describes the methodology to perform the detailed measurement of OS noise for a GPOS, Chapter 4 describes the evaluation of the address translation management for lightweight kernels.

Part III describes the issues of designing a specialized runtime system to improve a specific class of applications. Chapter 5 describes a runtime system and a programming model to efficiently execute irregular application on a commodity cluster.

Part IV shows that implementing optimization at user-level can require considerable programming effort, suggesting that the runtime should be used to implement this logic. Chapter 6 shows the potential hardware thread priorities to improve several target metrics for various applications on POWER5 and POWER6 processors. Chapter 7 how a radix sort kernel implemented for an emerging many-core architecture can be optimized using hardware features like data locality management and network-on-chip. The final Chapter concludes summarizing the work presented in this thesis and proposing some possible future work.

Chapter 2

Background: current trends in High Performance Computing

This chapter introduces concepts that will be further expanded in the following chapters. In particular, this chapter describes the current technology trends, the characteristics of future exascale and big-data applications and the system software enabling those application to scale on future systems.

2.1 Technology trends

The last two decades witnessed an exponential growth of extreme scale systems performance. As Moore's law states, transistor densities double every 18 months. Industry has been following Moore's law to deliver more powerful processors every year. Leveraging the Moore's law and employing large-scale parallelism, supercomputer performance managed to grow even faster than processor performance.

Figure 2.1 shows the exponential growth of supercomputing power as recorded by the TOP500 [10]. From the early nineties GFlop systems to current PFlop systems, the sustained growth rate of supercomputers performance has been two times the growth rate of Moore's law.

This impressive performance growth as been possible because of several concurrent factors, including: sustained increase of processor performance, reduction of network latency, increment of network and memory bandwidth. If this growth rate continues, the next generation of supercomputers should be able to deliver a performance in the order of 10^{18} floating point operations per seconds (ExaFlop). The design of a new generation of supercomputers, called *Exascale* generation, is the object of large

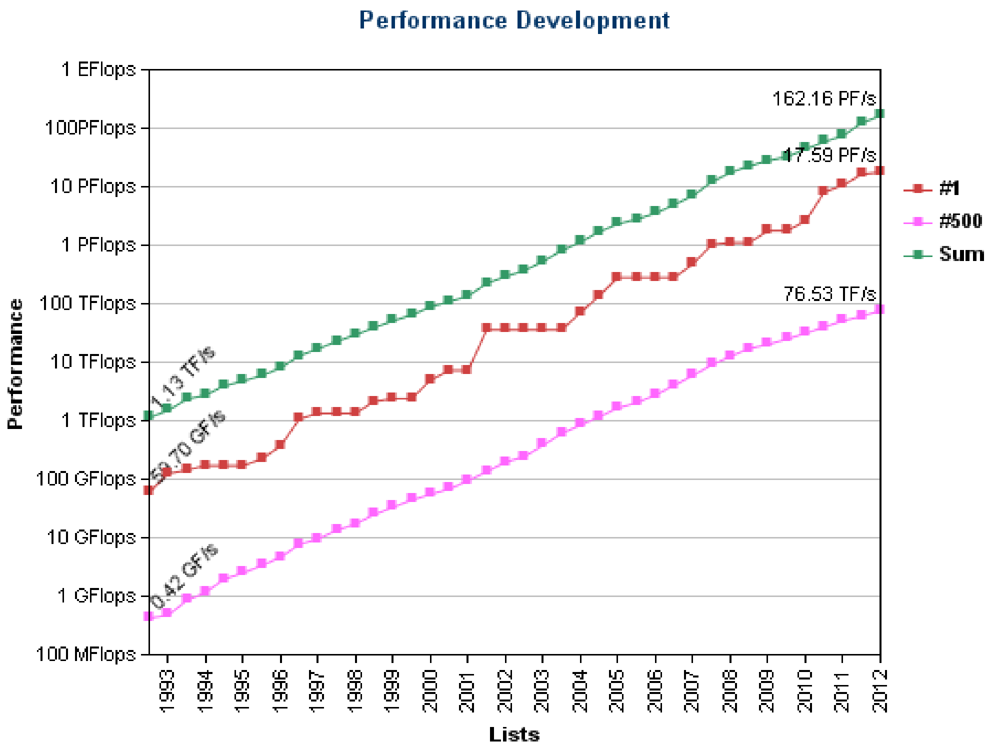


Figure 2.1: Exponential growth of supercomputing power as recorded by the TOP500 [10] - *#1* refers to the first in the list, *#500* refers to the 500th and *Sum* is the sum of all supercomputers in the list.

part of current research in HPC. The joint efforts of several institutions to design an Exascale supercomputer are the driving force behind many new development in computer architecture, operating systems, runtime systems, programming models, and large scale applications. Exascale systems will be the results of current technology trends meeting with application requirements, and leveraging the lessons learned from more than two decades of supercomputing. For this reason, observing the direction technology trends is a consistent and necessary approach to do sensible research in HPC. In the context of this research efforts, the Defense Advanced Research Projects Agency (DARPA) produced the first comprehensive study on the challenges in achieving HPC systems with ExaFlop performance [24]. The study concluded that there are four challenges to build exascale supercomputers:

1. **The Energy and Power Challenge**: Today it does not exist a technology able to deliver sufficiently powerful systems at connected wattages well below 100 MW. From a political-economic perspective has been suggested a power threshold of 20 MW, even though an exact power consumption constraint is not yet defined.

2. **The Memory and Storage Challenge**: There is a lack of mature memory and storage technologies that will be able to fulfil the I/O bandwidth requirements of future exascale systems within an acceptable power envelope.

3. **The Concurrency and Locality Challenge**: The end of increasing single thread processing performance and the trend towards many-core processing elements pose challenges to achieve the expected level of parallelism. Projections for exascale systems indicate that future applications may have to support more than a billion of separate threads in order to efficiently use the hardware.

4. **The Resilience Challenge**: This challenge is related to the explosive growth in the number of components in a supercomputer as well as the need to use advanced technology at extreme voltage and temperature operating points. Individual devices and circuits will become more and more sensitive to operating environments and hence resiliency and reliability will be of utmost importance.

The expected technology trends toward exascale computing is summarized in Table 2.1. As shown in Table 2.1, performance of various components of today extreme-scale systems are expected to grow at different rate. This is a major game changer for the design and implementation of extreme-scale systems, compared to today's systems. Dennard scaling states that while transistor size decreases (Moore's law) power density remains constant, so the transistor power usage decreases too. In the last decade

Metric	2012	2015	exascale	growth
System Peak [PF]	25	200	1000	100
Power [MW]	6-20	15-50	20-80	10
System Memory [PB]	0.3-0.5	5	32-64	100
Memory per Core [GB]	0.5-2	0.2-1	0.1-0.5	1/10
Node Perf. [GF]	160-10^3	500-7×10^4	10^3-10^4	10-10^3
Cores/Node	16-32	100-1,000	10^3-10^4	100 - 10^3
Node Mem. BW [GB/s]	70	100-10^3	400-4×10^3	100
Number of nodes	10^4-10^5	5×10^3-5×10^4	10^4-10^5	10-100
Total Concurrency	O(10^6)	O(10^7)	O(10^9)	10^4
MTTI	days	O(1 day)	O(1 day)	1/10

Table 2.1: Technology trends from current systems to ExaFlops systems [25]. Column growth refers to the expected growth from 2012 to exascale.

Dennard scaling ended and was evident that power density could not be maintained constant as transistor sizes were decreasing [64]. This implied a paradigm shift, because performance could not anymore be obtained with higher clock frequencies. Processor design started to change, from single core to multi and many core architectures. Because of this paradigm shift, modern applications need to exploit thread-level parallelism to compensate for the lack of single-core performance. New processors are expected to have hundreds to thousands cores and to leverage hardware multi-threading. As a consequence, the available memory per core is decreasing. While performance in older system was dependent mostly on computational power (i.e , FLOPS) this is not the case for exascale. Data-movement across the system, through the memory hierarchy and even for register-to-register operations is expected to be the main contributor to energy consumption.

Figure 2.2 shows the energy cost of data communication between several parts of the system. The cost of register access is lower than the cost of a double precision floating point operation. Nonetheless off-chip communications produce an increase in energy consumption of two orders of magnitude.

2.2 Scientific and large-scale applications

Scientific applications are becoming more and more vital to do research in a wide variety of scientific fields. Besides experimentation, simulation is today one of the fundamental methodologies of modern scientific research. Through simulation of physical systems scientists can validate hypothesis and make prediction with an increasing level of ac-

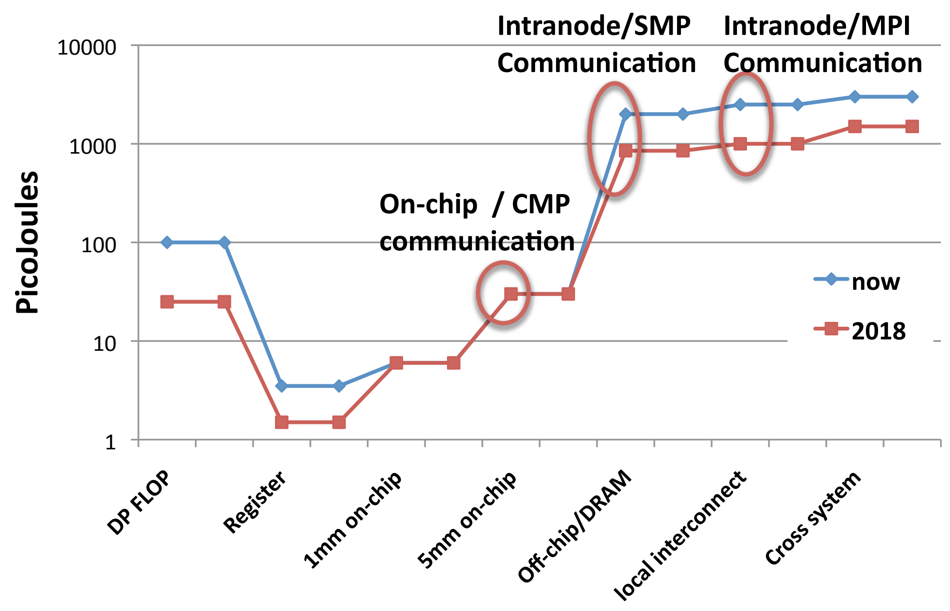


Figure 2.2: Energy consumption for data communication, difference between today and in 2018 [24]

curacy. Simulations are an opportunity not only for scientific research but also for a large number of industrial and service sectors such as Energy, Aeronautic, Mechanic, Finance and Health. Scientific applications are commonly developed by domain experts with a basic knowledge of the languages and programming models available on their systems. Most of the scientific applications end to be very complex codes, because of the necessity to solve complex scientific problems in the shortest possible time.

The European Exascale Software Initiative (ESSI) enumerates several grand challenges for exascale computing [25]. These challenges could improve the state of the art of many scientific medical and engineering problems. Some of the grand challenges identified by the ESSI are:

- **Therapy efficiency**: genetic sequencing is quickly becoming a primary tool to identify and prevent health risks for a growing number of diseases. The improvement of sequencing instruments by a factor of 10^3 to 10^6 it will make possible to integrate genomic data into clinical trials. This will have a huge impact on drug development and therapies. This will be possible only if exascale systems will enable the management of ExaBytes of sequencing data.

- **Extreme Weather Forecast**: Computational modeling is a necessary tool to integrate with physical earth observations to better understand the weather processes. The accuracy of computational models has a direct impact on the ability of predicting natural hazard and foster policy-making for risk-mitigation. Uncertainty quantification also plays a central role in evaluating the potential outcomes of extreme natural events. The economic, social and environmental impacts of this challenge are enormous.

- **Greening the Aircraft**: as energy is becoming a huge issue for every aspect of the social and economic life, future air transportation systems will have to reduce their energy consumption and their impact on the environment. These reductions include a reduction of emission by 50% and a decrease of external noise level by 10-20 dB (known as the vision-2020 plan).
- **Materials**: exascale is expected to bring major changes also in the Material science. Focus of simulations is expected to shift from a qualitative description of basic phenomena to the numerical optimization and quantitative analysis of soft material properties, micro-structural evolution and chemistry-driven problems.

2.2.1 Big data and exascale computing

In the 20th century, simulation has been referred as the *third paradigm* of scientific discovery, theory and experimentation being the first two. Large-scale scientific simulations have in fact enabled deep understanding of physical systems where experimentation is difficult, hazardous, or very expensive. Enabling this new paradigm of scientific discovery has fostered the design of state of the art HPC systems and software, driving research in computer science and engineering. The availability of large-scale models and simulation and the availability of large instruments like telescopes, colliders and light sources has generated a growing volume of scientific data. Computing resources to move, analyze and manage this exponentially growing volume of data is becoming the next big challenge [48], commonly referred as *big data*. The efforts to extract knowledge from large scientific dataset is considered today the *fourth paradigm* of scientific discovery. Because of the increasing velocity, heterogeneity, and volume of the data generated, extracting knowledge from large datasets is becoming a key aspect of science.

The steps toward the design of an Exascale Supercomputer are increasingly pointing to the necessity of integrating Exascale with big data. The integration of this two aspects of scientific discovery will be able to solve scientific problems orders of magnitude more complex than today. The Department of Energy (DOE) ASCAC Report for

Data Intensive computing identifies the typical knowledge discovery life-cycle as follows [48]:

1. **Generation**: Data is generated using simulation with large-scale computational facilities, sensors network and other type of high-throughput instruments. The output of this phase is a raw dataset in the order of PetaBytes up to ExaBytes.

2. **Processing and organization**: The data is then processed to be more usable. This phase includes, reorganization, filtering, subsets creation, distribution and several pre-processing methodologies. The output of this phase is a pre-processed, structured dataset.

3. **Analytics, Mining and Knowledge discovery**: In this phase domain-specific scalable algorithms explore the data to analyze patterns and statistics, extract and discover information or perform data-mining. The output of this phase is sensible domain-specific knowledge, usually reduced by several order of magnitudes with respect to the original dataset.

4. **Action, Feedback and Refinement**: In the last phase, the knowledge produced is used to close the loop validating or rejecting hypothesis, improve models or take effective actions in the specific domain.

The life-cycle described above is implemented through a large variety of applications, from the high end large-scale simulations to the data-mining and knowledge discovery algorithms. The critical aspect of applications working with big data is to have higher data access rates than traditional compute-intensive applications. For this reason, this type of applications are called *data-intensive* applications, i.e., applications where performance are dominated by data access rather than computation. To exemplify this type of applications Table 2.2 shows a comparison of performance metrics for compute-intensive and data-intensive benchmarks.

The benchmarks SPECINT and SPECFP [86] are used for integer and floating-point operations, respectively. MediaBench [106] is used for multimedia workloads. TPC-H [7] is a benchmark for decision support applications. MineBench is used as representative of data mining and analytics applications. The performance metrics shown in Table 2.2 shows that SPECINT SPECFP MediaBench and TPC-H have a different behavior with respect to MineBench. In particular, MineBench shows significantly more Data References than the others, resembling a characteristic data-intensive workload. Because of the data references, MineBench also experiences more bus accesses and L2

Parameter	SPECINT	SPECFP	MediaBench	TPC-H	MineBench
Data References	0.81	0.55	0.56	0.48	1.10
Bus Accesses	0.030	0.034	0.002	0.010	0.037
Instruction Decodes	1.17	1.02	1.28	1.08	0.78
Resource Related Stalls	0.66	1.04	0.14	0.69	0.43
ALU instructions	0.25	0.29	0.27	0.30	0.31
L1 misses	0.023	0.008	0.010	0.029	0.016
L2 Misses	0.0030	0.0030	0.0004	0.0020	0.0060
Branches	0.13	0.03	0.16	0.11	0.14
Branch misprediction	0.0090	0.0008	0.0160	0.0006	0.0060

Table 2.2: Performance metrics for compute-intensive and data-intensive benchmarks. Numbers shown are values per instruction.[48]

misses. A high rate of data references characterize data-intensive applications, quantifying the percentage of instructions that access in-memory data. Given the growing cost of data access outside the chip and, even higher, outside the local node, performance of applications with a high rate of data references are going to be dominated by data-access instructions.

2.2.2 Data-intensive and Irregular Applications

As explained in the previous sections, data-intensive applications are going to play a central role in scientific discovery in the next exascale systems. Besides being characterized for the high data access rate, data-intensive applications often show very poor data locality. The *locality principle* is a fundamental principle of computer architecture [85], stating that programs tend to reuse data and instructions they have used recently. Locality can mainly be of two different types: *temporal locality* and *spatial locality*: the former states that data used recently is likely to be reused, while the latter states that given a data access items which addresses are close are likely to be accessed soon.

Based on the degree of available locality in the data, applications can be divided in *regular applications* and *irregular applications*. Regular applications can be programmed to access data in an organized fashion, thus exploiting the high temporal and spatial locality available in their data. Typical examples of regular applications are programs based on dense vectors and matrices, as in matrix factorization and stencil computation. Given the regularity of such problems, the associated algorithms usually employ synchronized blocking collectives to exploit data parallelism.

On the other side of the spectrum we have the so-called irregular applications. Because of the unpredictable data access pattern, these applications have very poor spatial and temporal locality. Irregular applications, thus present unpredictable memory and network access patterns. Moreover, applications in this class often show an unpredictable instruction flow, reducing instructions locality and hence the ability to predict their behaviour.

Big-data and in general data-intensive applications often fall into this class. Typically, these are applications that exploit pointer-based data structures, such as linked-lists, graphs, unbalanced trees and unstructured grids. Applications of this class are graph exploration algorithms, data mining, social network analysis and, in general, tools related to knowledge discovery from large data sets. These accesses are usually fine-grained and highly unpredictable. Besides being very large, the datasets are usually very difficult to partition. Although these problems present high inherent parallelism, modern HPC systems are not optimized for them. Besides irregular memory and network access patterns, applications in this class often show an irregular instruction flow, reducing the source code locality and the ability to predict their behaviour.

In fact, the current large-scale systems focus around regular computation and data access patterns. They target flop intensive scientific applications that are easy to partition, present high arithmetic intensity and can exploit complex cache-based architectures to reduce data access latencies.

2.3 System software for large-scale systems

Even if research in system software has sometimes been marginalized, the difficulty of efficiently manage large-scale system resources has repeatedly shown the importance of providing scalable system software. Over the years, system software has evolved to keep the pace with technology and applications evolution. Research on scalable system software for large-scale system has historically been driven by two fundamental aspects: on one side, performance, efficiency and reliability, on the other side the goal of providing a productive developing environment. As for most of the engineering challenges, obtaining the right balance between two requirements such as performance and productivity is not trivial. Computer scientist working in this field have taken two different approaches: modifying the software stack used in high-end single node machines (such as high-throughput servers), and on the other hand, trying to develop part of the stack from scratch such as specialized runtime and operating systems. The trends in hardware technologies described in the previous sections will introduce additional complexity and the need for highly adaptable and scalable system software. Moreover, the difficulty of developing large-scale applications with an explicit message passing programming model (i.e., message passing interface - MPI) as it is commonly done today, will introduce newer programming models. This new developments imply radical changes at all levels of the system software stack, and in particular for the OS and the runtime system.

2.3.1 Operating Systems

The role of the Operating System (OS) is to manage hardware resources on behalf of the runtime system and the applications. The OS, because of its position at the bottom of the software stack (if we exclude the Hypervisor) requires upper layers such as runtime, libraries, and applications to be implemented using its API. For this reason, the development and success of a new OS depends not only on its inherent technical characteristics, but also on the availability of a rich software eco-system living on top of the new OS.

The choice of a General Purpose Operating System (GPOS), rather than a specialized OS for large-scale systems, has its advantages precisely in its software ecosystem and in the familiarity of applications developers with its API. The most used GPOS for large-scale system, Linux, can count on a wide-range of services and libraries for developing any kind of application, scientific applications included. Linux has been ported on most of the processor produced and provides a large hardware-driver base for any kind of device and component. Moreover, the Linux API is probably the most taught system software API at industrial and academic level, because of the advantages of the open source development philosophy. For all these reasons, Linux has naturally been the candidate GPOS to run on large-scale high-performance systems.

Figure 2.3 shows that first Unix-like OSes and nowadays Linux have been the most used operating systems for the supercomputers in the TOP500 list. As mentioned before, using GPOS for large-scale high performance systems has its drawbacks. When HPC systems approached larger scale, it became clear that predictable performance was one of the fundamental requisite. Linux, on the other hand, is not designed to provide a constant predictable execution time to the applications. Linux has a variety of system services (daemons), scheduling policies and interrupts that provide necessary services for desktop or server oriented applications. Those services can be a significant source of noise (so called OS-noise or jitter) and worsen performance predictability. A more important consequence of OS noise is observed for that applications based on periodic collective synchronizations (such as *MPI_Waitall*). Many of the currently used scientific large-scale applications (e.g. Sequoia benchmarks [110]) are organized in

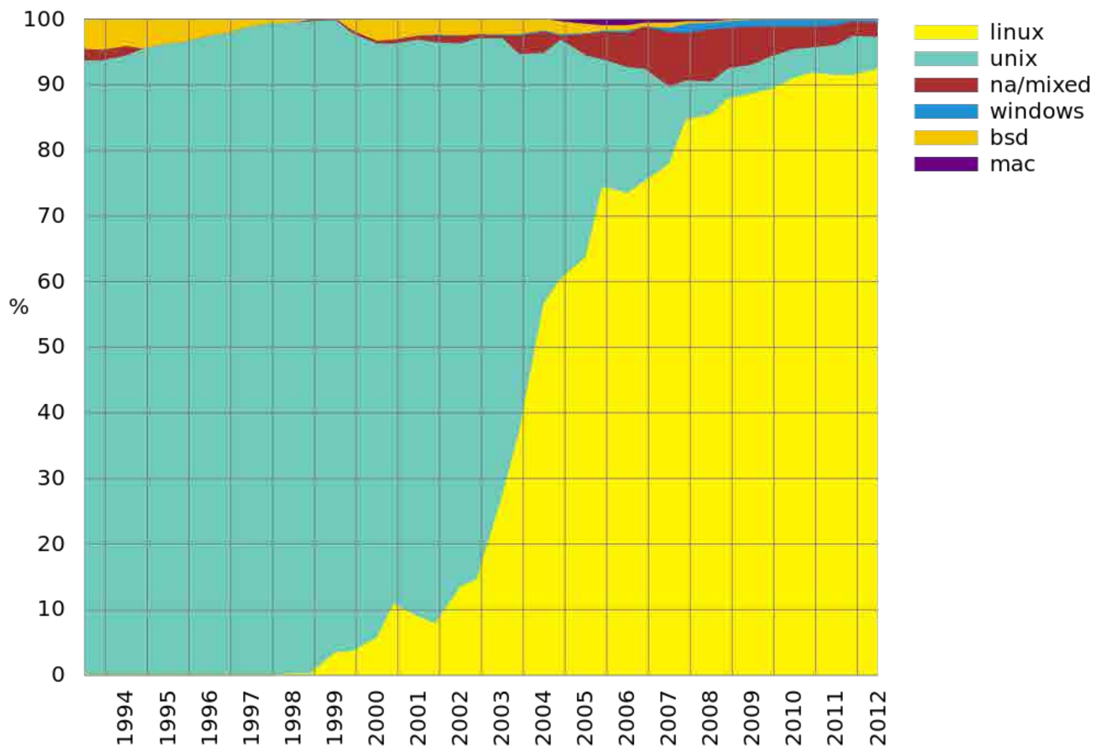


Figure 2.3: Percentage of the TOP500 using a given Operating System [10]

alternating computation and communication phases, and make an intensive use of this type of collectives. The delays introduced by the OS-noise in synchronized collectives may cause noise propagation that significantly increases execution time and reduces performance predictability. Figure 2.4 shows the trace of a parallel application with multiple threads. The computing phases (yellow) are followed by a communication phase (barrier in black). OS interruptions (in blue) can delay one thread forcing all the remaining threads to wait the delayed thread during the communication phase (barrier). This effect is called noise propagation (or amplification) and its frequency increases at scale because of the higher probability of an OS interruption on one of the parallel threads.

To reduce the overhead and complexity of unnecessary system services and to reduce OS noise, researchers have designed specialized lightweight OS for large-scale systems (lightweight kernels or microkernels). The first lightweight kernel was popularized with the development of Mach in 1985[79]. Then, lightweight kernels have been a constant presence in the space of system software for supercomputers, even if not the most common adopted solution. Today's Sandia's Kitten, IBM CNK and Cray CNL are three

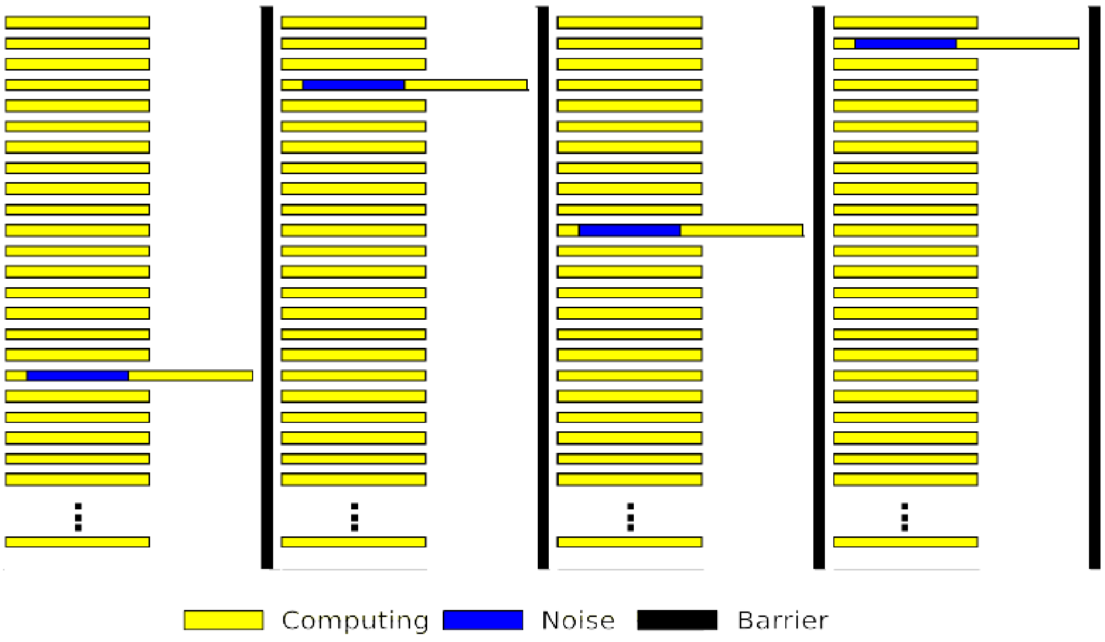


Figure 2.4: Effect of an OS interruption on a parallel application.

of the most well-known example of this approach running on modern supercomputers. In order to leverage the wide software base and the use of a familiar API, lightweight kernels tend to implement a subset of the standard Linux API. The advantages of using a lightweight kernel with respect to a GPOS are, between others, increased performance, efficiency and reliability.

2.3.2 Runtime Systems and Parallel Programming Models

The runtime system is the layer of the software stack between the OS and the applications. Its role is to act on behalf of the application to make the best use of the hardware resources exposed by the OS. With respect to the OS, the runtime system has specific knowledge of the application characteristics and requirements. Furthermore, runtime systems are often specialized to support applications with specific characteristics. The presence of a runtime system simplify the development of applications providing commonly used software abstractions that hide the complexity of the underling OS API.

The software abstractions implemented in the runtime system are at the core of the programming model used by the applications. Parallel programming models are the software abstractions used to express the parallelism of a program. The term parallel programming model is often used with different meanings: (a) to describe the abstract parallel machine, the basic operations performed on the machine and their semantic (b) to describe the actual implementation of a programming model, including programming languages, compiler directives, libraries and programming environments. It is sometimes difficult do draw a line between the abstract concepts composing a programming model and its actual implementation. We will refer to the meaning (a) as *program structure model and communication model* while to the meaning (b) as *programming models*.

Three factors are very useful when evaluating the optimal programming model for a given domain:

1. Perfomance: usually expressed in throughput or execution time.
2. Productivity: the advent of new kind of applications and the complexity of programming current many-core clusters is pointing to the need for high-productivity programming models.
3. Generality: the third factor is sometimes overlooked, but is of utmost importance when combined with the other two. In fact, reducing a programming model generality is the only effective approach to increase both performance and productivity.

One of the lessons learned in the last decade with the proliferation of parallel programming models is that the optimal programming model depends on the domain. An efficient approach is to employ specialized runtime systems for the specific domain of the problem, in order to obtain significant advantages in terms of performance and productivity.

Research on parallel programming models spans several decades and has generated a myriad of programming languages and libraries. The actual abstractions and ideas behind many implementations are often the same, or a combination of them. In the following, we describe some of the most well-known abstractions and ideas used in modern parallel programming models, the program structure model and the communication model.

Program Structure Model:

- *SPMD* - Single Program Multiple Data is a program structure composed of a fixed number of threads that run the same program on different sets of data. The threads are started and terminated synchronously.
- *Fork/Join* - a single thread of execution (main thread) is started and then several threads can be dynamically created from the main thread with the *fork* construct.

The main thread than waits for the children completion with a *join* construct. A Fork/Join model support nested parallelism if the fork construct can be recursively used by the children to create other threads.

- *Master/Worker* - a single master thread distribute the work to a fixed number of workers threads. This construct is commonly used to provide an efficient use of hardware resources (load balancing).
- *Loop Parallelism* - a *for* loop is used as basic construct to express parallelism. The iteration of the loop are executed in parallel on several threads. This model can support nested parallelism with the implementation of nested loops constructs.
- *Dataflow parallelism* - dataflow programming is a programming paradigm that models a program as a directed graph of the data flowing between operations, thus exploiting data parallelism and implementing dataflow principles and architecture [142] .

Communication Model:

- *Message Passing* - threads communicate with explicit exchange of messages. In a *two-sided* communication, the receiver executes a *receive* operation that is matched by a *send* operation of the sender. In a *one-sided* communication the sender thread executes a *put* operation to send a message on a predefined location that can be accessed by the receiver.
- *Shared Memory* - with a shared memory system different threads of execution can communicate through the memory. They communicate by reading and writing on the shared memory using the same memory locations.
- *Partitioned Global Address Space* - this model is the equivalent of a shared memory model for distributed memory systems (e.g. clusters). Since the underlying hardware does not have a single memory system shared among the threads, the programming model provide an abstraction for it. The abstraction is a global address space that can be accessed by all the threads. System memory is partitioned between local and global memory. Each thread has access to its own local memory and also to the global memory abstraction to communicate with the other threads.

Many programming models express parallelism with one or more program structure models and also more than one communication model (hybrid programming models).

Besides program structure and communication model there are other criteria that can be used to compare programming models[99]:

- *Implementation* - whether the programming model is implemented with a new programming language, extends a serial programming language (e.g. using compiler directives or adding keywords), or is implemented through programming libraries (API).

- *Worker management* - this criteria describes the creation and management of execution units whether they are tasks, threads or processes. Worker management is called *explicit* when the programmer needs to express the creation and termination of the execution units. Otherwise is called *implicit*.

- *Workload partitioning scheme* - worker partitioning is the way the workload is partitioned into smaller units of work called tasks. Worker partitioning is *explicit* when the programmer needs to express how the work is partitioned, while is *implicit* when the programmer only has to specify a section of code to be parallel.

- *Task-to-worker mapping* - it defines how the units of work are mapped into the workers. It can be *explicit* when the programmer has to manually define the mapping, and *implicit* when the programming model automatically provides the mapping.

- *Synchronization* - it defines the order in which the workers access shared data. With *explicit* synchronization the programmer has to express the access order with synchronization constructs, while with *implicit* synchronization the programming model does it transparently.

The data locality can be managed by the programmer to efficiently exploit local memory. We survey some of the most relevant parallel programming models for HPC (alphabetical order):

Active Pebbles [166] is lightweight data-drive programming language based on active messages [164]. Active pebbles combines fork/join program structure with a PGAS communication model. It also supports messages coalescing and active routing to speedup the communication.

CUDA is a parallel programming model developed by NVIDIA [133]. The CUDA runtime is designed to scale transparently with the growing number of cores found in modern GPUs. CUDA allows the programmer to use an high-level C programming language. The CUDA programming model exploits data parallelism and is particularly effective for regular floating-point intensive computations.

Chapel [46] (Cascade High Productivity Language) is a general-purpose parallel programming language developed by Cray, Inc. under the Darpa High Productivity Computing Systems (HPCS) program. Chapel has been developed for general parallel programming, locality aware programming, object oriented programming and generic programming [50].

Charm++ [96] is a parallel programming system based on a message-driven migratable-objects programming model. The Charm++ runtime consist of a number of medium-grained processes that communicate with messages. Charm++ supports communication latency tolerance through suspending the blocking process and executing a waiting process.

Cilk[31] is a task-based implementation of a shared memory model trough work-stealing . This approach is particularly good for highly recursive parallel problems. It is implemented as a C/C++ extension and does not support distributed memory systems, but can be easily combined with MPI.

Coarray Fortran [131] is a fortran extension for parallel systems. It features a SPMD programming structure and provides constructs to specify the data distribution through a PGAS model.

Global Arrays [129] provides a SPMD programming model with a PGAS communication model. The data locality can be managed by the programmer to efficiently exploit local memory. Globaly Arrays is composed of a number of processes sharing a global address space and communicating with one-sided primitives.

Habanero-Java [41] is an extension of the original Java definition of the X10 language. The Habanero-Java extension is primarily focused on task parallelism and supports asynchronouse creation of tasks with a fork/join program structure model and a PGAS communication model.

MPI [121] (Message Passing Interface) can be considered the most widely used programming model for distributed memory systems. It is a pure message passing programming model with an SPMD program structure. MPI is a specification for message passing operations. Various MPI implementations provide libraries with one-sided and two-sided communication mechanism to send messages to different nodes in a cluster. The programmer has to explicitly specify each communication message between processes. The execution threads (processes) do not share any data structure and communicate explicitly through messages. Even if the use of explicit messages has advantages for the application scalability, explicitly controlling the communication for large parallel applications is extremely complex. Moreover, with MPI alone, SMP systems with shared memory cannot benefit from the ability to share data structures in the same node. Hybrid programming models use MPI for inter-node communication and a shared memory programming model for intra-node communication.

OmpSs [37] is an extension of OpenMP to integrate features from the StarSs [103] programming model. OmpSs enable asynchronous parallelism by the use of data-dependencies between different tasks of the program. Tasks can be created dinamically and the runtime transparently manages data-dependencies and task scheduling.

OpenCL [101] is a parallel open-standard programming model similar to CUDA. However, OpenCL is more general-purpose than CUDA and can be used on a variety of heterogeneous architectures including CPUs and GPUs. Unlike CUDA, that is primarily focused on data parallelism, OpenCL it is also designed to support task parallelism.

OpenMP [134] is on the other hand the most widely used shared memory programming model. It provides program structures such as fork/join and loop parallelism and is implemented both as language extension and programming library. OpenMP it's easier to program than many other approaches but is restricted to a single multi-core or many-core system.

ParalleX (PX) [70] is an experimental execution model that exploits a split-phase multithreaded transaction distributed computing methodology that decuples computation and communication [70]. ParalleX supports fine-grain multithreading, global address space, overlapping of communication and computation and is able to move work to the data. High Performance PX (HPX) is a runtime system based on the parallex execution model.

Pthreads [38] is a POSIX implementation of a thread library that enables explicit control of each execution thread and shared memory access. Fine grain control of the thread life-cycle introduces a higher level of complexity. It is often used as building block for programming model with a higher level of abstractions.

SHMEM [136] allows developers to write a parallel application using a globally accessible shared memory abstraction. The key features of SHMEM are one-sided point-to-point communications, a shared memory view, and atomic operations on globally visible variables.

Titanium [170] is a Java-derived language and system for HPC. It provides a SPMD program structure with a global address space. An important feature of Titanium is the possibility of running unmodified on uniprocessors, shared memory machines and distributed memory systems.

Unified Parallel C (UPC) [60] is a programming model that employs a global address space abstraction partitioned between all the nodes. UPC is an extension of the C language designed for HPC systems. UPC uses the SPMD programming structure and SPMD communication model.

X10 [47] is a Java-derived, object-oriented parallel language developed by IBM. X10 was designed to enable scalability, high-performance and high productivity. It is based on a PGAS model and implements the Globally Asynchronous Locally Synchronous model (GALS) originally developed for embedded hardware [123].

2.3.3 Trends in System Software

The ongoing changes in large-scale and high-performance technology and applications are forcing traditional system software to renew. Next generation system software for HPC is expected to provide system services with improved performance and power efficiency. Currently, several trends are emerging in system software design.

One of the clear requirements is that next-generation system software has to be low-overhead. Asynchronous OS noise and high system software overhead will not be acceptable in an exascale system. For this reason, current GPOS are expected to be heavily redesigned in order to reduce overhead. The availability of a large number of cores can be exploited to completely separate the execution of system software code from the application code. Application and system software executions will be

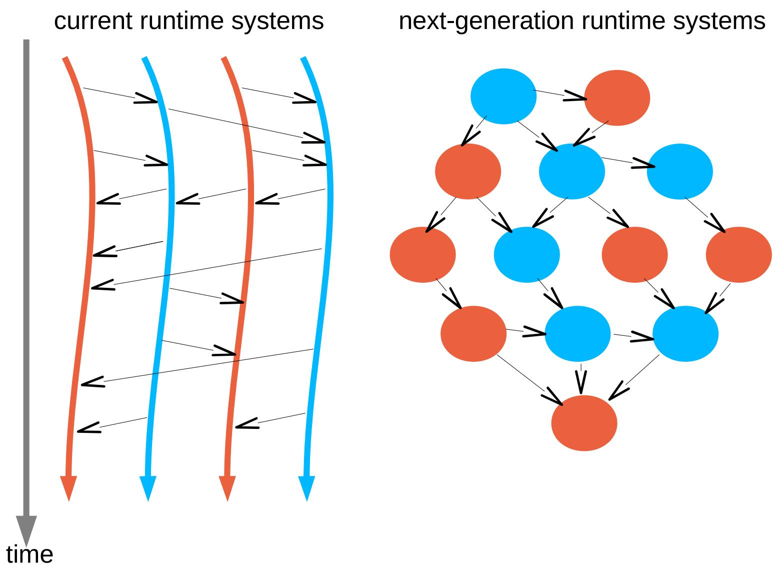


Figure 2.5: Programming model schema for current and next-generation runtime systems.

separated by space (different cores) rather than time (time-sharing). Moreover, heterogeneous architectures with specialized cores for different workloads will allow higher performance and power efficiency to execute application and system software code.

Another big change will be in the level of parallelism that system software is expected to manage. Current programming models are based on the idea of communicating processes (CSP) while next-generation will be a fine-grained event-driven and adaptive programming model. Current generation runtime systems are implemented with a number of parallel processes that progress through execution exchanging (mostly) synchronous messages. On the other side, in the next-generation of runtime systems there will be a large number of small tasks communicating with each other with asynchronous. Figure 2.5 shows a graphical representation of current and next-generation programming models. If we look at new runtime systems such as Chapel, HPX, and X10, and to architectural supports as TERAFLUX [77] we observe that they are in fact developed according to this principles.

Given the expected changes in technology and programming models for the future generation of supercomputers, the runtime system is expected to play a central role towards the design of a scalable software stack.

In fact, because of the need for a lightweight OS, several services will be moved into the runtime system. The runtime system has the advantage of a much closer knowledge about the running application. Services such as scheduling, memory, locality management and memory object synchronization can be more efficiently handled by the runtime system.

Another important trend is that to obtain high power efficiency the system software will feature a much higher level of adaptivity. Based on the application behavior, it will be necessary to adapt the hardware (e.g. turning off cores, dynamic voltage and frequency scaling, power capping) in order to save energy. To obtain adaptivity, a higher level of logic will be included at the runtime system level in order to increase the ability to take resource management decisions while the application is running.

All the mentioned trends point out the fact that the runtime system will be one of the most important areas for improvements for next-generation large-scale systems.

Part II

Operating System Scalability

Chapter 3

General Purpose Operating Systems

This part of the thesis addresses the problem of operating system scalability. The OS often plays a fundamental role in the scalability of large parallel systems. Generally, operating systems for large scale machines fall in two categories: general purpose OS (e.g. Linux) and lightweight OS (CNK, Catamount etc.). In this chapter, we focus on the effect of using a general purpose operating system on a large-scale high-performance system. As mentioned before, general purpose OS offer the advantage of a well-known programming environment, plenty of features and a large code-base. On the other hand, they also present scalability issues. One of the most studied scalability issues is the so-called OS noise (or Jitter). In the following sections, we present a methodology to analyze the scalability of general purpose operating systems through detailed measurement of OS noise on real HPC applications.

3.1 Summary

Operating system (OS) noise (or jitter) is a well-known problem in the High Performance Computing (HPC) community. Petrini et al.[137, 68] showed how OS noise may limit application's scalability, severely reducing performance on large scale machines and large system activities (e.g., network file system server). Later studies[68, 148] confirmed the early conclusions on different systems and identified timer interrupts and process preemption as main sources of OS noise[75, 156]. Most of those studies are "qualitative" in that they do a good job characterizing the overall effect of OS noise on the scalability of parallel applications, but tend not to identify and characterize each kernel noise event. For example, many studies showed that the timer interrupt is an important source of OS noise, but few of them provided information about which activities are executed during each timer interrupt. Such information would be helpful to developers trying to reduce OS noise. For example, operating systems use timer interrupts to periodically start several kinds of activities, such as bookkeeping, watchdogs, page reclaiming, scheduling, or drivers' bottom halves. Each activity has a specific frequency and its duration varies according to the amount of work to process, and thus different timer interrupts can introduce different amounts of OS noise. Our detailed quantitative analysis allows developers to differentiate and characterize the noise induced by each event triggered on an interrupt and thus address the pertinent sources.

Lightweight and micro kernels are common approaches taken to reduce OS noise on HPC systems. Many supercomputers have run a lightweight kernel, such as Compute Node Kernel (CNK)[71] or Catamount[140]. Lightweight kernels provide functionality for a targeted set of applications (for example, HPC or embedded applications) and usually introduce negligible noise. They usually do not take periodic timer interrupts or TLB misses, but typically provide restricted scheduling and dynamic memory capability.

The lightweight approach worked well in the past when HPC applications only required scientific libraries and a message passing interface (MPI) implementation[121, 69]. The NAS benchmark suite[124] contains examples of these kind of applications. Also, the requirements of many other of the largest applications, such as previous Gordon Bell winners[39, 17], most of the Sequoia benchmark suite[110], and others[5, 2] can be satisfied by this approach.

However, modern HPC applications are becoming more complex, such as UMT from the LLNL Sequoia benchmarks[110], Climate code[100], and UQ[23]. Moreover, in order to improve performance there is a growing interest in richer shared-memory programming models, e.g., OpenMP[134], PGAS, such as UPC, Charm++, etc. At the same time, dynamic libraries, python scripts, dynamic memory allocation, checkpoint-restart systems, tracers to collect performance and debugging information, and virtualization[104], are seeing wider use in HPC systems. All these technologies require richer support from the OS and the system software. Moreover, applications being run on supercomputers are no longer coming strictly from classical scientific domains but from new fields, such as financial, data analytics, and recognition, mining, and synthesis (RMS), etc. Current trends indicate large petascale to exascale systems are going to desire a richer system software ecosystem.

Possible solutions to support such complexity include: 1) extending lightweight and micro kernels to provide support for modern applications needs as mentioned above; 2) tailoring a general purpose OS, such as Linux or Solaris, to the HPC domain (e.g., the Cray Linux Environment for Cray XT machines); and 3) running a general purpose OS in a virtualized environment (e.g., Palacios[104]). All the three scenarios above have the potential to induce more noise than yesterday's lightweight kernels. Thus, it will become more important to be able to accurately identify and characterize noise induced by the system software.

We have developed a technique to provide a quantitative descriptive analysis for each event of OS noise. The mechanism allows us to detail all sources of OS noise through precise kernel instrumentation and provides frequency and duration analysis for each event. In addition to providing a much richer set of data, we have integrated this collected data with the Paraver[138] visualization tool allowing developers to more easily analyze the data and get an intuitive sense for its implications. Paraver is a classical tool for parallel application's performance analysis, and integrated with our data can provide a view of the OS noise introduced throughout the execution of HPC applications.

Overall, in this chapter we describe the following three contributions: First, we describe a methodology whereby detailed quantitative information may be obtained for each OS noise event. As mentioned our methodology is most useful for HPC OS designers and kernel developers trying to provide a system well suited to run HPC applications. Though not the thrust of this work, we show how we implemented that methodology by augmenting Linux Trace Toolkit Next Generation (LTTng) [55, 56]. We validate our approach by comparing it to other well-known standard techniques to analyze OS noise, such FTQ (Fixed Time Quantum). Second, we provide a case study in which we use our methodology to analyze the OS noise when running benchmarks from the LLNL Sequoia applications. Although our methodology and tools may well be useful for application programmers, in this work we focus on providing the insights that can be gained at the system level. Thus, while we do run real applications, it is not for the purpose of studying those applications, but rather for demonstrating that our methodology allows us to obtain a better understanding of the system while running real applications. Our experiments enrich and expand previous results with our quantitative characterization. Third, we describe how this detailed characterization allows us to disambiguate noise signatures of qualitatively similar events allowing developers to address the true cause of a given noise event.

3.2 Experimental environment

Table 7.1 shows the experimental environment used in this chapter. The table includes the hardware system, the operating system, the communication library and the benchmarks used for the experiments.

CPU	Dual Quad-Core AMD Opteron(tm) processor workstation at 2.1 GHz (2 sockets - 4 cores per socket)
Memory	64 Gigabyte of RAM
Network	Ethernet (used only to support NFS)
System size	single node
Operating System	Linux 2.6.33.4
Communication Library	OpenMPI
Benchmarks	Sequoia benchmarks[110] (AMG, IRS, LAMMPS, SPHOT, UMT)

Table 3.1: Experimental environment

3.3 Related Work

Operating system noise and its impact on parallel applications has been extensively studied via various techniques, including noise injection [68]. In the HPC community, Petrini et al. [137] explained how OS noise and other system activities, not necessarily at the OS level, could dramatically impact the performance of a large cluster. In the same work, they observed that the impact of the system noise when scaling on 8K processors was large due to *noise resonance* and that leaving one processor idle to take care of the system activities led to a performance improvement of 1.87×. Though the authors did not identify each source of OS noise, a following paper [75] identified timer interrupts, and the activities started by the paired interrupt handler, as the main source of OS noise (*micro OS noise*). Nataraj et al. [125], also describe a technology to measure kernel events and make this information available to application-level performance measurements tools.

Others [156, 53] found the same result using different methodologies. The work of De et al. [53] is the most similar to ours. They perform an analysis of the OS noise instrumenting the functions `do_IRQ` and `schedule`, and obtaining interrupts and process preemptions statistics. Our approach is similar, but we instrumented all kernel entry points and activities, thus providing a complete OS noise analysis. As a consequence, we are able to measure events like page faults and softirqs that significantly contribute to OS noise.

The other major cause of OS noise is the scheduler. The kernel can swap HPC processes out in order to run other processes, including kernel daemons. This problem has also been extensively studied [137, 75, 156, 53, 95, 74], and several solutions are available [155, 74]. The effects on multiprocessor memory subsystem of three causes of OS noise (kernel memory references, process scheduling, virtual-to-physical address translation) have been also evaluated through the use of real and synthetic traces [76]: such approach allows for faster evaluations.

Studies group OS noise into two categories of high-frequency, short-duration noise (e.g. timer interrupts) and low-frequency, long-duration noise (e.g. kernel threads) [68]. Impact on HPC applications is higher when the OS noise resonates with the application, so that high-frequency, fine-grained noise affects more fine-grained applications, and low-frequency, coarse-grained noise affects more coarse-grained applications [137, 68].

The impact of the operating system on classical MPI operations, such as collective, is examined in Beckman et al. [22].

There are several examples of operating systems in the literature designed for HPC applications. Solutions can be divided into three classes: micro kernels, lightweight kernels (LWKs), and monolithic kernels. *L4* [109] is a family of micro-kernels designed and updated to achieve high independence from the platform and improve security, isolation, and robustness. The *Exokernel* [62, 61, 63] was developed with the idea that the OS should act as an executive for small programs provided by the application software, so the Exokernel only guarantees that these programs use the hardware safely. *K42* [16] is a high performance, open source, general purpose research operating system kernel for cache-coherent multiprocessors that was designed to scale to hundreds of processors and to address the reliability and fault-tolerance requirements of large commercial applications. Sandia National Laboratories have also developed LWKs with predictable performance and scalability as major goals for HPC systems [140].

IBM *Compute Node Kernel* for Blue Gene supercomputers [71, 119] is an example of a lightweight OS targeted for HPC systems. CNK is a standard open-source, vendor-supported, OS that provides maximum performance and scales to hundreds of thousands of nodes. CNK is a lightweight kernel that provides only the services required by HPC applications with a focus of providing maximum performance. Besides CNK, which runs on the compute nodes, Blue Gene solutions use other operating systems. The I/O nodes, which use the same processors as the compute nodes, run a modified version of Linux that includes a special kernel daemon (*CIOD*) that handle I/O operations, and front-end and service nodes that run a classical Linux OS [119].

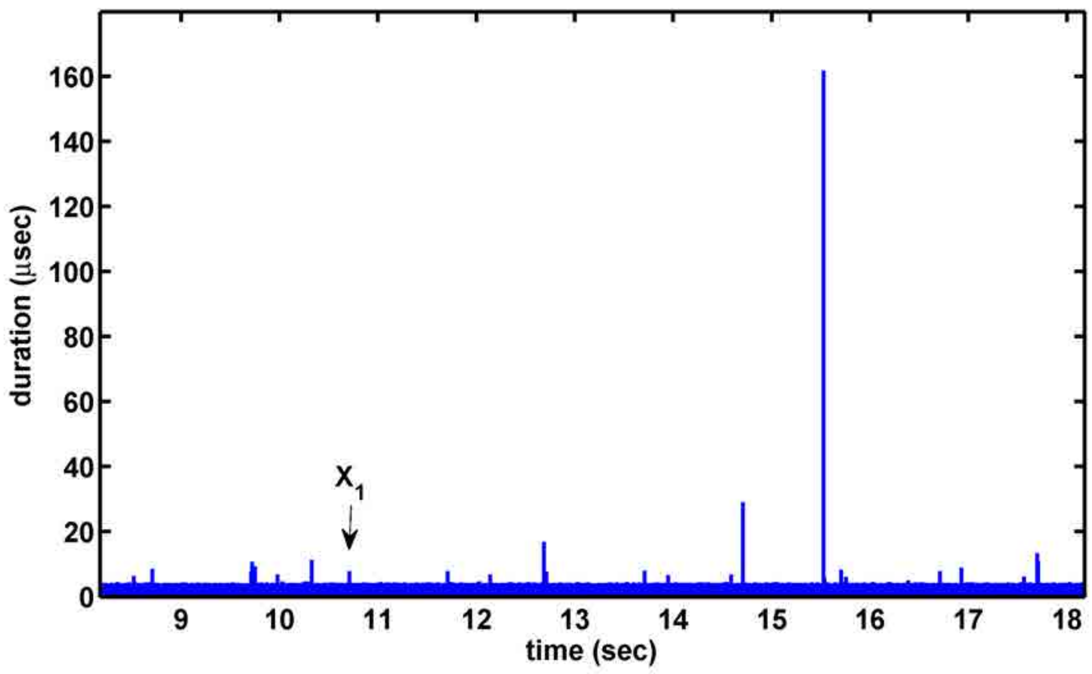
Lightweight and micro kernels usually partner with library services, often in user space, to perform traditional system functionality, either for design or performance reasons. CNK maps hardware into the user's address space allowing services such as messaging to occur, for efficiency reasons, in user space. The disadvantages of CNK come from its specialized implementation. It provides a limited number of processes/threads, minimal dynamic memory support, and contains no `fork`/`exec` support[71]. Moreira et al. [119] show two cases (socket I/O and support for Multiple Program Multiple Data (MPMD) applications) where IBM had to extend CNK in order to satisfy the requirements of a particular customer. Overall, experience shows that this situations are quite common with micro and lightweight kernels. There are several variants of full weight kernels that have customized general-purpose OSes. ZeptoOS [19, 20, 21] is an alternative, Linux-based OS, available for Blue Gene machines. ZeptOS aims to provide the performance of CNK while providing increased Linux compatibility. Jones et al. [95] provides a detailed explanation of the modification introduced to IBM AIX to reduce the execution time of MPI `Allreduce`. They showed that some of the OS activities were easy to remove while others required AIX modifications. The authors prioritize HPC tasks over user and kernel daemons by playing with the process priority and alternating periods of favored and unfavored priority. HPL[74] reduces OS noise introduced by the scheduler by prioritizing HPC processes over user and kernel daemons and by avoid unnecessary CPU migrations.

Shmueli et al. [148] provide a comparison between CNK and Linux on BlueGene/L, showing that one of the main items limiting Linux scalability on BlueGene/L is the high number of TLB misses. Although the authors do not address scheduling, by using the HugeTLB library, they achieve scalability comparable to CNK (although not with the same performance).

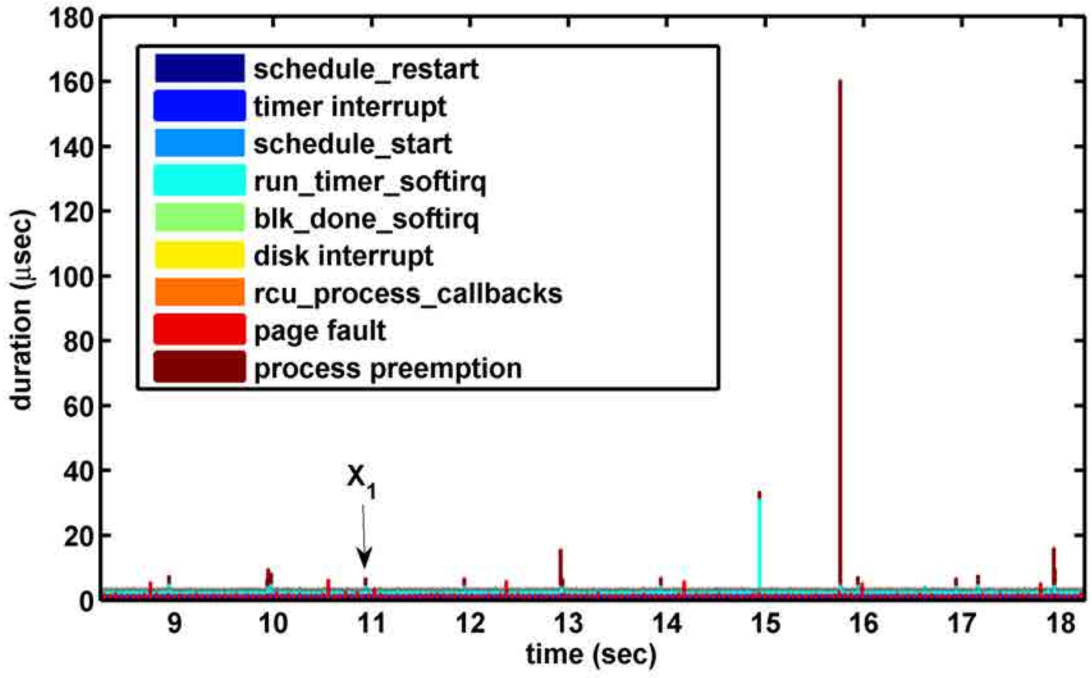
Mann and Mittal [54] use the secondary hardware thread of IBM POWER5 and POWER6 processors to handle OS noise introduced by Linux. They reduce OS noise by employing Linux features such as the real time scheduler, process priority, and interrupt redirection. The OS noise reduction comes at the cost of losing the computing power of the second hardware thread. Mann and Mittal consider SMT interference a source of OS noise.

3.4 Measuring OS noise

The usual way to measure OS noise on a compute node consists of running micro benchmarks with a known and highly predictable amount of computation per time interval (or quantum), and measuring the divergence in each interval from the amount of time



(a) OS noise as measured by FTQ



(b) Synthetic OS noise chart

Figure 3.1: Measuring OS noise using FTQ and LTTng-noise. Figures 3.1a and 3.2a report the direct OS overhead obtained multiplying the execution time of a basic operation by the number of missing operations.

taken to perform that computation. An example of such as technique is the *Finite Time Quantum* (FTQ) benchmark proposed by Sottile and Minnich[154]. FTQ measures the amount of work done in a fixed time quantum in terms of *basic operations*. In each time interval T, the benchmark tracks how many basic operations were performed. Let N_{max} be the maximum number of basic operations that can be performed in a time interval T. Then we can indirectly estimate the amount of OS noise, in terms of basic operations, from the difference $N_{max} - N_i$, where N_i is the number of basic operations performed during the i-th interval.

Figure 3.1a shows the output of FTQ on our test machine. Each spike represents the amount of time the machine was running kernel code instead of FTQ (obtained multiplying the number of missing basic operations by the time required to perform a basic operation). Whether the kernel interrupted FTQ one, two, or more times, and what the kernel did during each interruption is not reported.[1] This imprecision is one disadvantage of indirect external approaches to measuring OS noise. An advantage of this approach is that experiments are simple and provide quick relative comparisons between different versions as developers work on reducing noise.

As we stated earlier, our goal was to provide a more detailed *quantitative* evaluation of each OS noise event. In order to obtain this information in Linux, instrumentation points includes all the kernel entry and exit points (interrupts, system calls, exceptions, etc.), and the main OS functions (such as the scheduler, softirqs, or memory management[36]). To obtain a detailed trace of OS events, we extended the LTTng with 1) an infrastructure to analyze and covert the LTTng output into formats useful to analyze OS noise, and 2) extra instrumentation points inside the Linux kernel. We call this extended LTTng infrastructure LTTng-noise.

The output from LTTng-noise includes a *Synthetic OS Noise Chart* and an *OS Noise Trace*. The former is a graph similar to the one generated by FTQ and provides a view of the amount of noise introduced by the OS. The latter is an execution trace of the application that shows all kernel activities.

Figure 3.1b shows the Synthetic OS Noise Chart for FTQ generated by LTTng-noise (as shown in Section 3.6, we can obtain similar graphs for any application). The OS Noise Trace can be visualized with standard trace visualizers commonly used for HPC application performance analysis. The current implementation of LTTng-noise supports the Paraver[138] trace format, but other formats can be generated relatively easily by performing a different offline transformation of the original trace file.

[1]It is possible to guess that the small, frequent spikes are related to the timer interrupt, as was pointed out by previous work[75, 156, 26].

3.4.1 LTTng-noise

A concern about LTTng-noise was the overhead introduced by the instrumentation. If the overhead is too large, the instrumentation may change the characteristics of the applications making the analysis invalid. Minimizing introduced noise, or more accurately keeping it below a measureable level, was especially critical for us as we were specifically trying to measure noise. In addition, HPC applications are susceptible to this induced noise causing an additive detrimental effect. Fortunately, LTTng was designed with similar goals [55], and our kernel modifications do not add extra overhead. In particular, LTTng has been designed with the following properties: 1) low system overhead, 2) high-scalability for multi-core architectures, and 3) high-precision. Low system overhead is obtained with a pre-processing approach, i.e., the kernel is instrumented statically and data are analyzed offline. High-scalability is ensured by the use of per-cpu data and by employing lock-less tracing mechanisms[55]. Finally, high-precision is obtained using the CPU timestamp counter providing a time granularity on the order of nanoseconds. The results of the low-overhead design of LTTng and our careful modification is an overhead in the order of 0.28% in average (measured among all the LLNL Sequoia applications we tested).

Since it is not possible to know in advance which kernel activities affect HPC applications, we collected all possible information. LTTng already provides a wide coverage of the Linux kernel. Even with these capability, it is still important to determine which events are needed for the noise analysis and, eventually add new trace points. To this end, we located all the entry and exit points and identified the components and section of code of interest. We then modified LTTng by adding extra information to existing trace points and new instrumentation points, for those components that were not already instrumented.

Another important consideration is which kernel activities should be considered noise (i.e., timer interrupts) and which are, instead, services required by applications and should be considered part of the normal application's execution (i.e., a `read` system call). We consider OS noise all those activities that are not explicitly requested by the applications but that are necessary for the correct functioning of the compute node. Timer interrupts, scheduling, page faults are examples of such activities. Second, we only account kernel activities as introduced noise when an application's process is runnable. During the offline OS noise analysis, we do not consider a kernel interruption as noise if, when it occurs, a process is blocked waiting for communication.

Even with these modifications the number of kernel activities tracked is considerable. Since it is not possible to show all the tracked activities, we focus on those that appear more often or have large impact (such as timer interrupts, scheduling, and page faults). However, developers concerned about specific areas can use our infrastructure to drill down into any particular area of interest by simply applying different filters.[1]

The second extension we made to LTTng is an offline trace transformation tool. We developed an external LTTng module that generates execution traces suitable for Paraver[138]. Additionally, the module generate a data format that can be used as input to Matlab. We use this to derive the synthetic OS noise chart and the other graphs presented in this chapter. We took particular care of nested events, i.e., events that happen while the OS is already performing other activities. For example, the local timer may raise an interrupt while the kernel is performing a tasklet. Handling nested events is particularly important for obtaining correct statistics.

3.4.2 Tracing scalability

The amount of data generated to trace OS events on a single node is not an issue. On the other hand, the application of any trace methodology to clusters composed by a large number of nodes (i.e. thousands of nodes), face the challenge of collecting and storing a very large amount of data at run-time. This problem arises for any approach willing to capture OS noise events with a fine granularity in HPC systems.

Given that OS noise is inherently redundant across nodes, one of the most effective solution is to enable tracing only on a statistically significant subset of the cluster's nodes. Another option is to apply data-compression techniques at run-time to reduce the data-size.

3.4.3 Analyzing FTQ with LTTng-noise

A comparison of Figure 3.1a and Figure 3.1b shows that LTTNG-NOISE captures the OS noise identified by FTQ. The small differences between the two graphs can be explained by realizing that FTQ computes missing integral number of basic operations and multiplying this by the cost of each operation thus producing discretized values, while LTTNG-NOISE calculates the measured OS noise between given points using the trace events. In general, the result is that FTQ slightly overestimates the OS noise, for FTQ does not account for partially completed basic operations.

Apart from these small differences, the figures show that the data output from these

[1]Filters are common features for HPC performance analysis tools and we provide the same capability for our Matlab module.

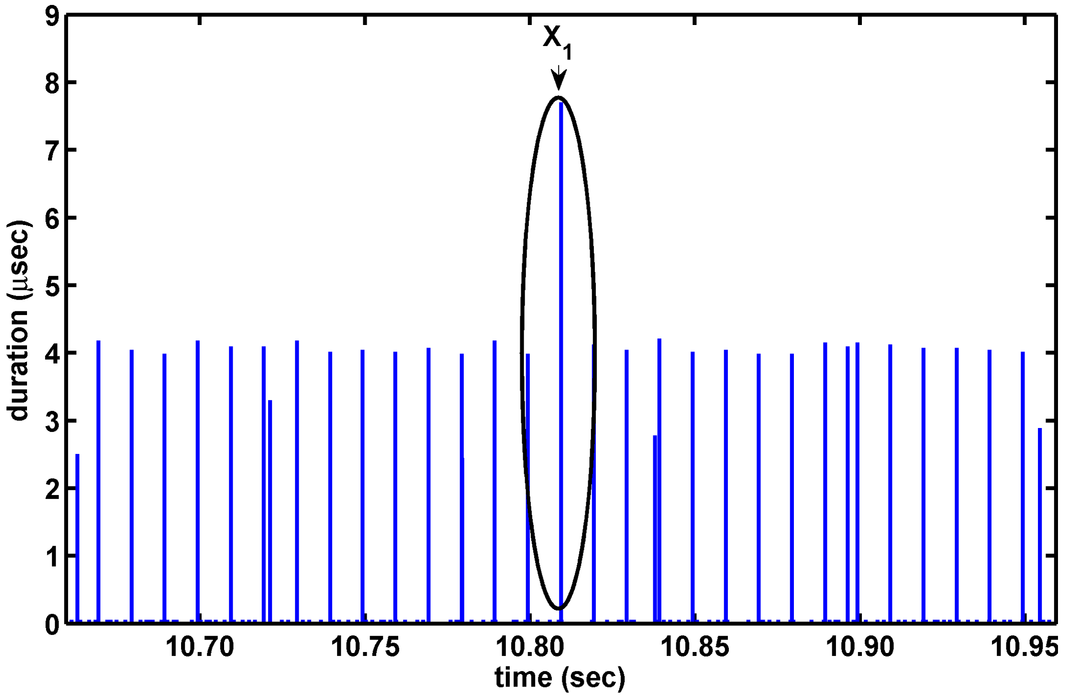
two methods are very similar. Thus, we have a high degree of confidence in comparing results from the different techniques, and therefore that our new technique represents an accurate view of induced OS noise. Unlike FTQ, our new technique allows quantifying and providing details for what contributed to the noise of each interruption.

The Synthetic OS noise chart in Figure 3.1b shows, for each OS interruption, the kernel activities performed and their durations. For example, at time $x = X_1$ FTQ detects 7.70 μsec OS overhead (this point is showed in Figure 3.1a). The corresponding point in Figure 3.1b (also highlighted on the chart) shows an overhead of 6.96 μsec but also shows that the interruption consists of `timer_interrupt` handler (1), a `run_timer_softirq` softirq (2), and a process preemption (`eventd` daemon) (3).

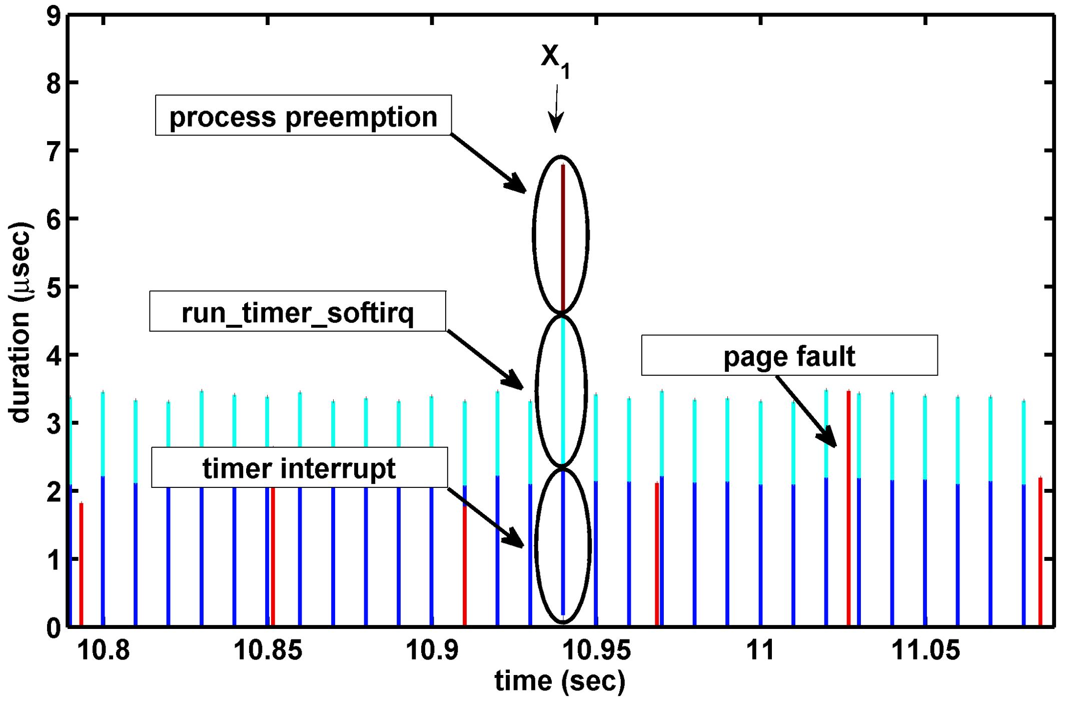
Figures 3.2a and 3.2b show details of Figures 3.1a and 3.1b, respectively, centered around point $x = X_1$. Figure 3.2b shows that, indeed, the small frequent spikes are related to the timer interrupt handler but, also, to the `run_timer_softirq` softirq, which takes about the same amount of time. Figure 3.2b also shows that there are smaller spikes, to the best of our knowledge not identified in previous work, that are similar to those generated by the timer interrupt handler, but that are not related to timers or other periodic activity. These interruptions are caused by page faults, which are, as we will see in Section 4.6, application-dependent.

The synthetic OS noise chart is useful to analyze the OS noise experienced by one process. HPC applications, however, consist of processes and threads distributed across nodes. Execution traces can be used to provide a deeper understanding of the relationships among the different application's processes. Moreover, performance analysis tools, such as Paraver, are able to quickly extract statistics, zoom on interesting portion of the application, remove or mask states, etc. Figures 3.3a and 3.3b show a portion of the execution trace of FTQ. The execution traces generated with LTTng-noise are very dense, even for short applications. Even if performance analysis tools are able to extract statistics and provide 3D views, the visual representation, especially for long applications, is often complex and lossy due to the number of pixels on a screen and the amount of information to be displayed at each time interval. For this reason, we usually zoom-in to show small, interesting portions of execution traces that highlight specific information or behavior.

Figure 3.3a shows 75 $msec$ of FTQ execution. In this trace white represents the application running in user mode, while the other colors represent different kernel activities. The trace shows the periodic timer interrupts (black lines) and the frequent page fault (red lines). While, in this case, page faults take approximately the same time, timer interrupts may trigger other activities (e.g., softirqs, scheduling, or process

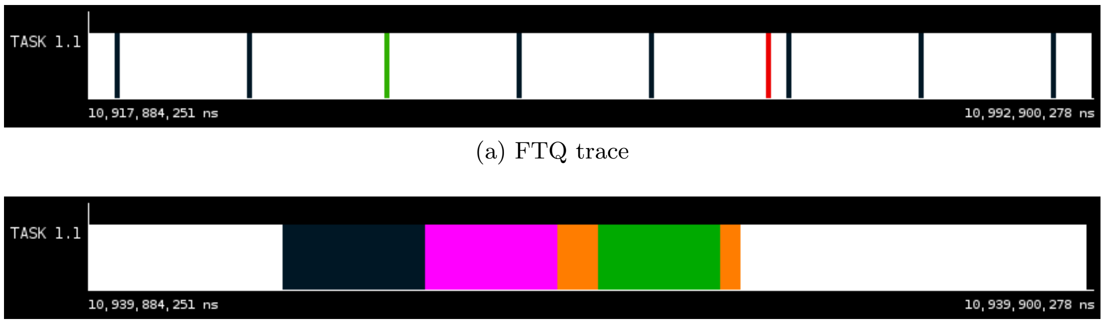


(a) OS noise as measured by FTQ (zoom)



(b) Synthetic OS noise chart (zoom)

Figure 3.2: Zoom on noise peak using FTQ and LTTng-noise. Figures 3.1b and 3.2b report time as measured by LTTng-noise.



(a) FTQ trace

(b) Zoom of the FTQ trace

Figure 3.3: FTQ Execution Trace. Figure 3.3a shows a part (75 *ms*) of the FTQ trace that highlights the periodic timer interrupts (black lines), the page faults (red line), and a process preemption (green line). Figure 3.3b zooms in and shows that the interruption consists of several kernel events. At this level of detail we can distinguish the timer interrupt (2.178 μsec, black) followed by the `run_timer_softirq` softirq (1.842 μsec, pink), the first part of the `schedule` (0.382μsec, orange), the process preemption (2.215 μsec, green), and the second part of the `schedule` (0.179μsec, orange).

preemption) and, thus, have different sizes. In order to better visualize the activities that were performed on a giving kernel interruption, Figure 3.3b shows a detail of the previous picture. In particular, Figure 3.3b depicts one interruption caused by a timer interrupt. Paraver provides several informations for each event, such as time duration or internal status, simply by clicking on the desired event. Regarding the detailed execution trace of FTQ, the tool provides the following time durations for each interruption: 2.178 μsec (timer interrupt), 1.842 μsec (`run_timer_softirq`), 0.382 μsec (first part of the `schedule` that leads to a process preemption), 2.215 μsec (process preemption), and 0.179 μsec (second part of the `schedule` that resumes the FTQ process).

3.5 Experimental Results

In this section we present our experimental results based on LTTng-noise and the Sequoia benchmarks[110] (AMG, IRS, LAMMPS, SPHOT, UMT). We ran experiments on a dual Quad-Core AMD Opteron(tm) processor workstation (for a total of 8 cores) with 64 Gigabyte of RAM. We used a standard Linux Kernel version 2.6.33.4 patched with LTTng-noise. In order to minimize noise activity and to perform representative experiments, we removed from the test machine all kernel and user daemons that do not usually run on a HPC compute node. The system is in a private network, isolated from the outside world, that consists of the machine itself and an NFS server. Most of the I/O operations, including the application's executable files and the input sets, are performed through the network interface, as it is often the case for HPC compute nodes. In order to obtain representative results, Sequoia benchmarks were configured to run with 8 MPI tasks (one task per core), and to last several minutes.

Section 3.5.1 analyzes the OS noise breakdown for each application. Sections 3.5.2, 3.5.3, 3.5.4, and 3.5.5 analyze in detail the OS activities identified by our infrastructure as the major source of OS noise. In each sub-section we provide synthetic statistical data and a few examples of more detailed information for chosen interesting applications.

3.5.1 Noise breakdown

Each application experience a different amount of OS noise. Table 3.2 summarizes the total time, the application time, the OS noise time and the percentage of OS noise.

Table 3.2: Application and Noise time measured in CPU cycles

	Total	Application	Noise	% Noise
AMG	900,436,075,066	892,327,859,639	8,108,215,427	0.90%
IRS	706,800,813,275	699,373,261,625	7,427,551,650	1.05%
LAMMPS	1,044,575,042,629	1,036,931,614,406	7,643,428,223	0.73%
SPHOT	693,157,887,616	692,839,730,149	318,157,467	0.05%
UMT	2,340,648,466,685	2,297,049,612,303	43,598,854,382	1.86%

Each OS interruption may consists of several different kernel activities. Our analysis allows us to break down each OS interruption into kernel activities. For simplicity and clarity, we focus on the kernel activities with larger contribution to OS noise. To this extent, we classified kernel activities into 5 categories:

periodic: timer interrupt handler and `run_timer_softirq`, the softirq responsible to execute expired software timers.

page fault: page fault exception handler.

scheduling: the `schedule` function and the related softirqs (`rcu_process_callbacks` and `run_rebalance_domains`).

preemption: kernel and user daemons that preempt the application's processes.

i/o: network interrupt handler, softirqs and tasklets.

Figure 3.4 shows the OS noise breakdown for the Sequoia applications. Each Sequoia application experiences OS noise in a different way. For example, page faults represent a large portion of OS noise for AMG and UMT (82.4% and 86.7%, respectively), while

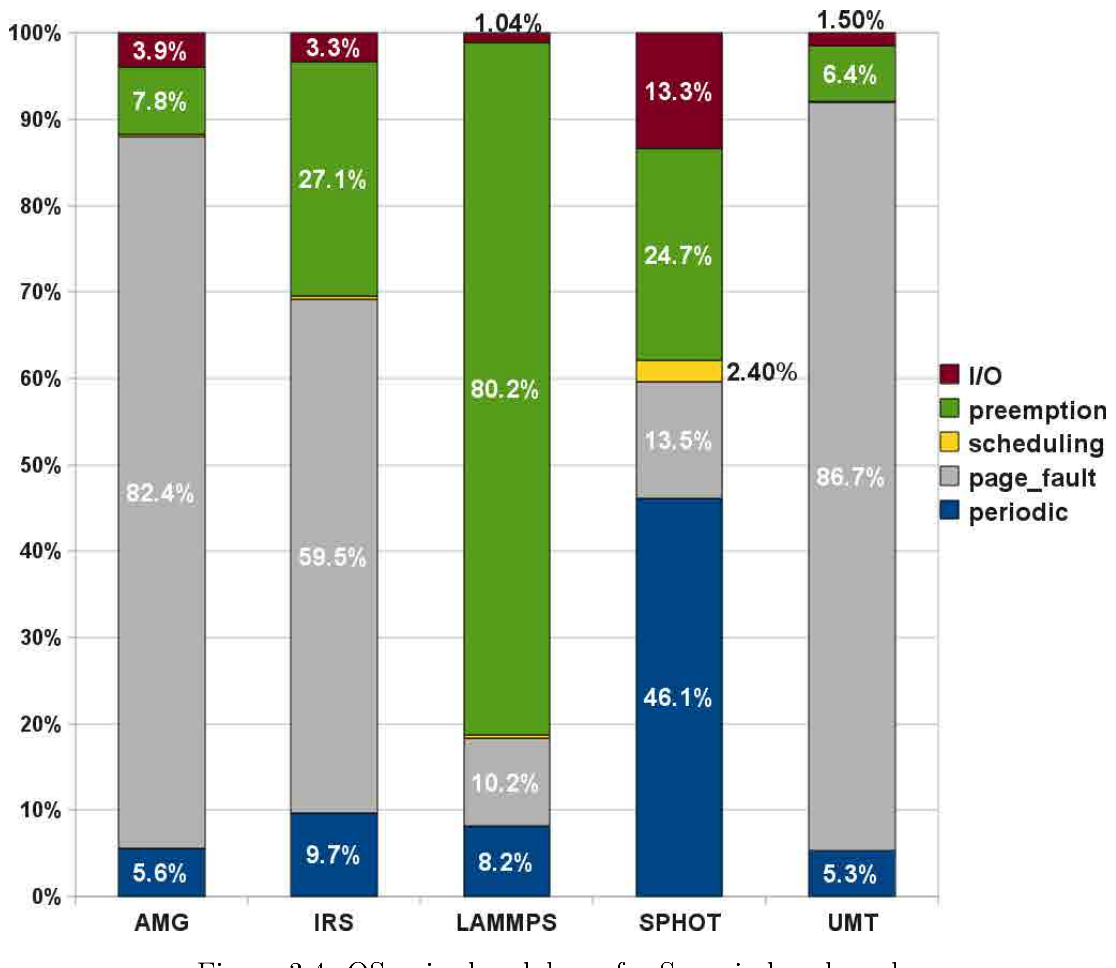


Figure 3.4: OS noise breakdown for Sequoia benchmarks

SPHOT and LAMMPS are only marginally affected by the same kernel activity (13.5% and 10.2%). IRS, SPHOT and, especially, LAMMPS are preempted by other processes during their computation, which represent a considerable component of their total jitter (27.1%, 24.7%, and 80.2%, respectively). Our analysis shows that the applications were interrupted particularly by `rpciod`, a I/O kernel daemon. Periodic activities (timer interrupt handler and periodic software timers) are, instead, limited (between 5% and 10%) for all applications but SPHOT.

3.5.2 Page faults

Page faults can be one of the largest source of OS noise. Memory management functionalities like page-on-demand and copy on write can significantly reduce the impact of page faults. Moreover, one of the main advantages of lightweight kernels is to drastically simplify dynamic memory management, thus reducing or even removing (as in CNK [71]) the noise caused by page faults.

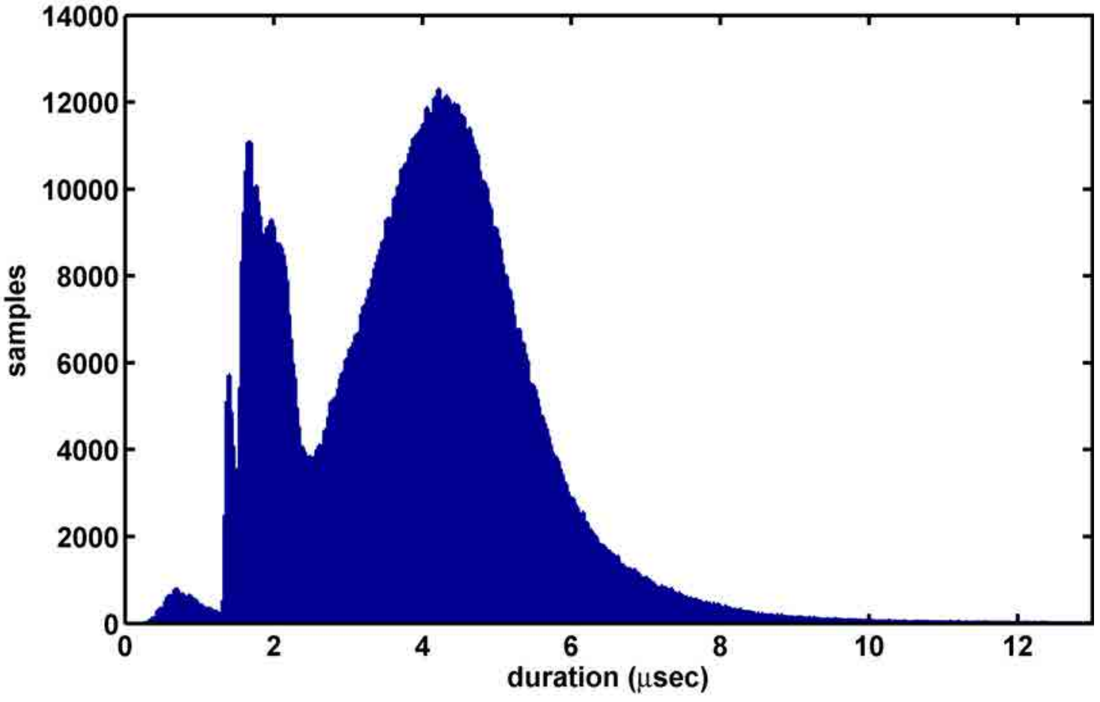
Table 3.3: Page fault statistics

	freq(ev/sec)	avg(nsec)	max(nsec)	min(nsec)
AMG	1,693	4,380	69,398,061	250
IRS	1,488	4,202	4,825,103	218
LAMMPS	231	3,221	27,544	248
SPHOT	25	2,467	889,333	221
UMT	3,554	4,545	50,208	229

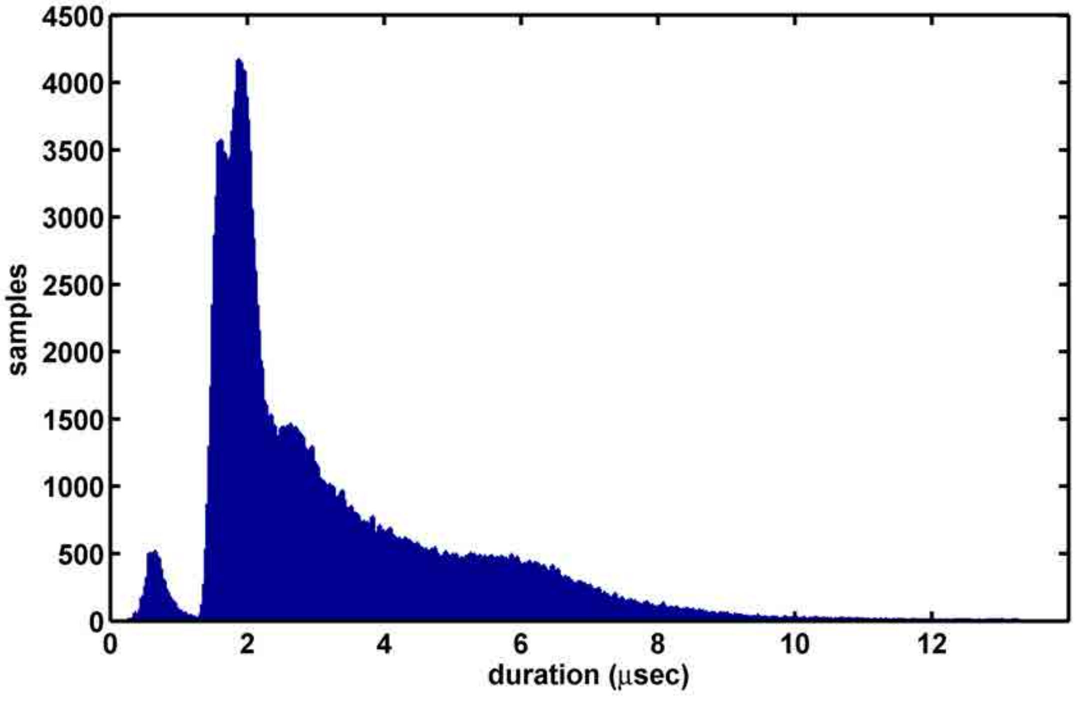
As Table 3.3 shows, for some applications, like AMG, IRS, and UMT, the frequency of page faults is even higher than that of the timer interrupt (10kHz in our configuration). Even more important is that the time series is very large and differs from application to application. From Table 3.3, we can observe that each application presents a different number of page faults. Moreover, though the minimum duration of a fault is similar across the benchmarks (about 250 *nsec*), the maximum duration varies from application to application (from 25.7 μsec to 69,398 μsec). OS noise activities that vary so much may limit application scalability on large machines, as shown in previous work [137, 75].

Figures 3.5a and 3.5b[1] show the page fault time series for AMG and LAMMPS, respectively. We chose these two applications because they present different distributions. Figure 3.5a (AMG) shows two main picks (around 2.5 μsec and 4.5 μsec), with

[1]Time distributions may have a very long tail that could make visualization difficult. To improve the visualization, we cut all the distributions in the histograms at the 99^o percentile.

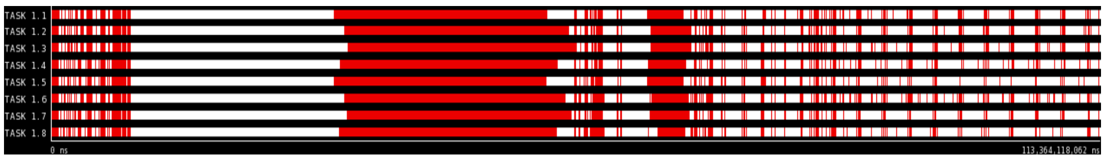


(a) AMG

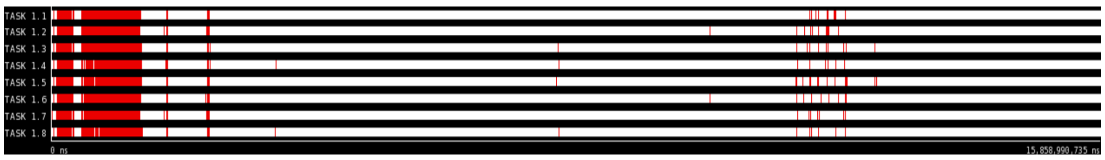


(b) LAMMPS

Figure 3.5: Page fault time series



(a) AMG



(b) LAMMPS

Figure 3.6: Page Fault Trace. Figures 3.6a and 3.6b show the execution trace of AMG and LAMMPS, respectively. We filtered out all the events but the page faults (red). The traces highlight the different distributions of page fault for AMG (throughout all the execution) adn LAMMPS (mainly located at the beginning and the end). Notice that in some regions page faults are very dense and appear as one large page fault when, in fact, there are thousands very close to each other but the pixel resolution does not allow distinguishing them.

a long tail. LAMMPS, on the other hand, shows a one-sided distribution with a main pick around 2.5 $\mu sec.$

Figure 3.6 shows another important difference between AMG and LAMMPS. While for LAMMPS page faults, marked in red in the pictures, are mainly located at the beginning (initialization phase), AMG page faults are spread throughout the whole execution, with several accumulation points. From these execution traces we can conclude that page faults are not of main concerns for LAMMPS, neither in terms of OS noise overhead (10.2% of the total OS noise, as reported in Figure 3.4) nor from the time series, as they mainly appear during the initialization phase. For AMG, instead, page faults may seriously affect performance. They represent a considerable portion of the total OS noise (82.4%), and may interrupt application's computing phases.

3.5.3 Scheduling

Scheduling activities are not only related to the `schedule` function but also to other kernel daemons and softirqs that are responsible, among other tasks, to keep the system balanced.

Our analysis shows that, indeed, the overhead introduced by the `schedule` function is negligible and constant, confirming the effectiveness of the new Completely Fair Scheduler (CFS) [118], which has a $O(1)$ complexity.

Domain balancing, instead, may create several problems in terms of execution time variability and, therefore, scalability[74]. In this section we analyze the effect of the `run_rebalance_domains` softirq. `run_rebalance_domains` is triggered when needed from the scheduler tick and it is in charge of checking whether the system is balanced and, if not, move tasks from one CPU to another. Domain rebalance is described in[74, 34, 36]. Here it is worthwhile to notice that rebalancing domains introduces two kinds of overheads: *direct* (time required to execute the rebalance code) and *indirect* (moving a process to another CPU may require extra time to warm up the local cache and other processor resources).

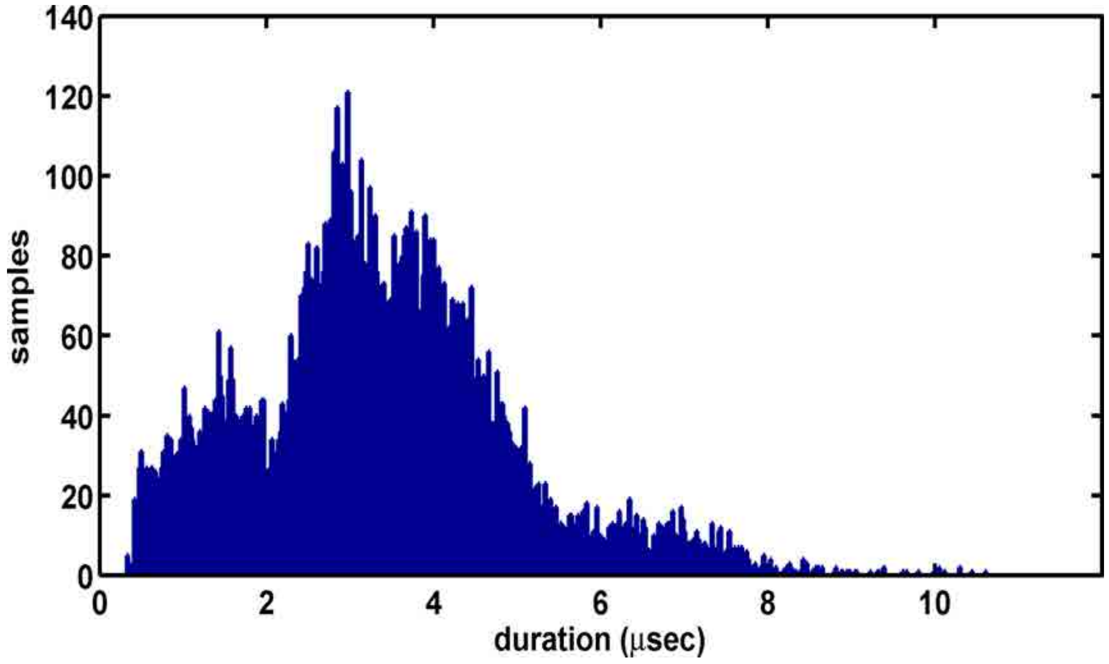
Figures 3.7a and 3.7b show the execution time series of the `run_rebalance_domains` softirq for UMT and IRS. While IRS shows a fairly compact distribution with a main pick around 1.80 μsec, UMT shows a much larger distribution with average of 3.36 μsec. Indeed, UMT is a complex application that involves, besides MPI, Python and pyMPI scripts. The OS has, thus, a much tougher job to balance UMT than IRS.

3.5.4 Process preemption and i/o

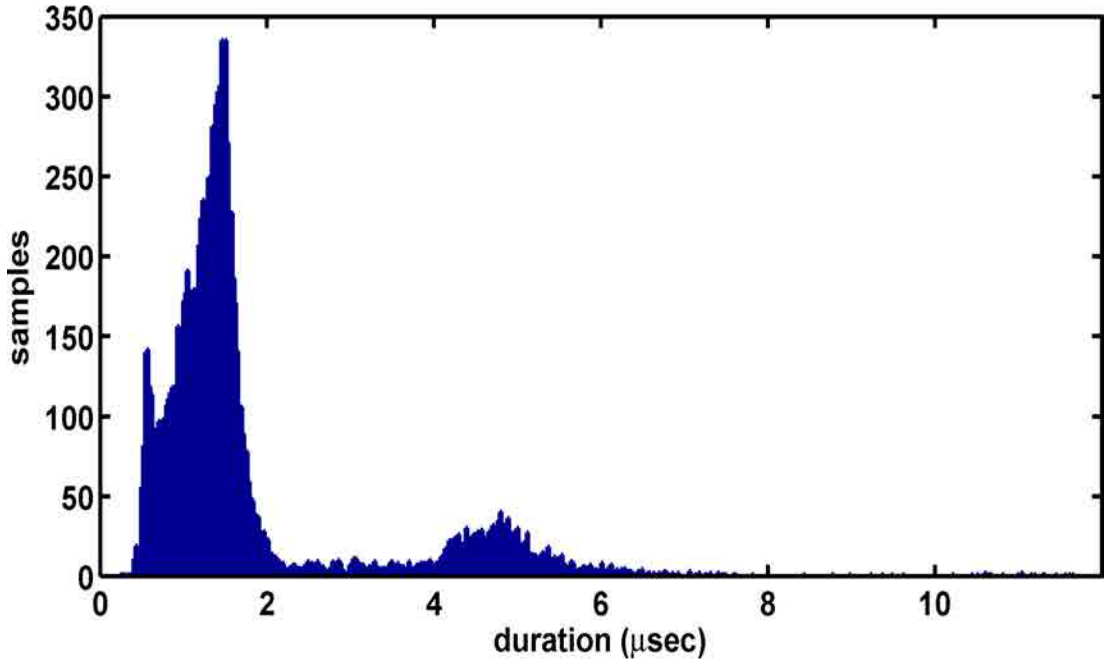
The OS scheduler may decide to suspend one or more running processes during the execution of a parallel application (*process preemption*). The suspended process is not able to progress during that time and the whole parallel application may be delayed. Typically, the OS suspends a process because there is another higher-priority process that needs to be executed. Kernel and user daemons are classical examples of such processes. The interrupted process may, in turn, be migrated by the scheduler on another CPU. This approach provides advantages if the target CPU is idle but may also force time sharing with another process of the parallel application. A discussion of the possible effect of process migration and preemption is provided in[74].

In our experiments the only kernel daemon that is very active is the I/O daemon (`rpciod`).[1] HPC compute nodes do not usually have disks or other I/O devices except, of course, the network adapter. Most HPC networks (such as Infiniband[146] or BlueGene Torus[12]) allow user applications to send/receive messages without involving the OS, through kernel bypass. All I/O operations (e.g., reading input data or writing final results) are shipped to an I/O node through the network. Once the operation is completed, the I/O node returns the results to the compute node.

[1]LTTng-noise also uses a kernel daemon to collect data at run time. This daemon is not supposed to be running in a normal environment, therefore we are not taking it into account in the rest of the discussion. However, we noticed that, though it depends on the application, the LTTng-noise kernel daemon was especially active at the beginning and at the endof the applictions. Moreover, as reported in section 3.4, the overall overhead of LTTng-noise is quite small.



(a) UMT



(b) IRS

Figure 3.7: Domain rebalance softirq time series

In our test environment, the compute node is connected to an NFS server through the `rpciod` I/O daemon. For most of the applications, `rpciod` is the only kernel daemon that generate OS noise. UMT is a different case because the application is more complex than the others. In particular, UMT runs several Python processes that may 1) interrupt the computing tasks, and 2) trigger process migration and domain balancing.

Figure 3.4 shows that the OS noise experienced by LAMMPS is dominated by process preemption, marked in green. Figure 3.8 shows that, indeed, LAMMPS processes are preempted several times throughout the execution. As opposed to the other benchmarks, LAMMPS performs a considerable amount of I/O operations. Since data are moved to/from the network, the OS suspends LAMMPS processes that issued I/O operations until the data tranfer is completed. In Linux, network I/O operations involve the network interrupt handler and the receiver (`net_rx_action`) and transmission (`net_tx_action`) tasklets.[1] On data transfer completion, the active network tasklet wakes up the suspended processes in the order I/O operations complete and on the CPU that receives the network interrupt. Clearly, that CPU may be running another LAMMPS process (which is preempted) at that time, thus a load balance reschedule may be triggered to move the preempted process on an idle CPU (migration).

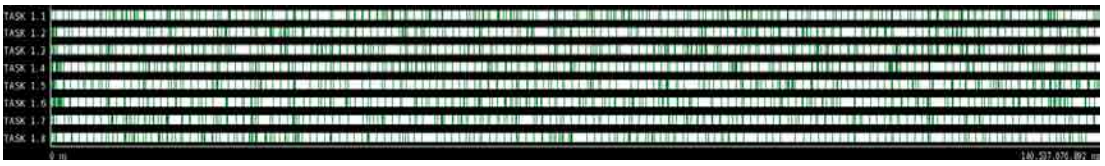


Figure 3.8: Process preemption experienced by LAMMPS. This picture shows the complete execution trace of LAMMPS. We filtered out all events but process preemptions (green). Though the pixel resolution does not allow distinguishing all the process preemption, it is clear that LAMMPS suffers many frequent preemptions.

Table 3.4: Network interrupt events frequency and duration

	freq(ev/sec)	avg(nsec)	max(nsec)	min(nsec)
AMG	116	1,552	347,902	540
IRS	87	1,666	353,294	521
LAMMPS	11	2,520	356,380	594
SPHOT	21	1,372	341,003	535
UMT	77	1,975	349,288	484

[1]Tasklets are implemented on top of softirqs and differ from the latter in that tasklets of the same type are always serialized. In other words, two tasklets of the same type cannot run on two different CPUs while two softirqs of the same type are allowed to do so[36].

Table 3.5: `net_rx_action` frequency and duration

	freq(ev/sec)	avg(nsec)	max(nsec)	min(nsec)
AMG	53	3,031	98,570	192
IRS	43	4,460	78,236	174
LAMMPS	10	4,707	84,152	199
SPHOT	15	1,987	45,150	207
UMT	22	5,484	75,042	167

Table 3.6: `net_tx_action` frequency and duration

	freq(ev/sec)	avg(nsec)	max(nsec)	min(nsec)
AMG	15	471	8,227	176
IRS	10	504	4,725	176
LAMMPS	2	559	4,392	175
SPHOT	3	409	2,746	200
UMT	9	545	8,902	173

Tables 3.4, 3.5 and 3.6 report the frequency and time duration of the network interrupt handler, the `net_rx_action`, and the `net_tx_action` tasklet, respectively. As we can see, the transmission tasklet is faster and more constant than the receiver tasklet. The reason is that while sending data to the NFS server is an asynchronous operation, receiving data from the NFS server must be done synchronously. In fact, the transmission tasklet can return right after the network DMA engine has been started, because the copy of the data from memory to the network adapter's buffer will be performed asynchronously by the DMA engine.

3.5.5 Periodic activities

Some activities are periodically started by the timer interrupt. With the introduction of high resolution timers in Linux 2.6.18, the local timer may raise an interrupt any time a high resolution timer expires. One of such timer is the *periodic timer interrupt*, which is in charge of accounting for the current process elapsed time and, eventually, call the scheduler if the process time quantum has expired. This interrupt is a well-known source of OS noise[75, 156]. In our test machine we set the frequency of this periodic high resolution timer to the lowest possible (100 Hz) so to minimize the effect of the periodic timer interrupt.

The term *timer interrupt* is often used to identify both the timer interrupt and the softirq. With our methodology, instead, we are able to distinguish between the timer interrupt (or *timer interrupt top half*) and the sotfirq `run_timer_softirq` (called *bottom half* in old Linux releases). The `run_timer_softirq`, in particular, may vary considerably across applications and between two separate instances of the event. This softirq, in fact, is in charge of running all the handlers connected to expired software timers. Each handler may have a different duration and applications may set software timers accordingly to their needs (for example, to take regular check points). It follows, that this activity may show a considerable execution time variation.

Figures 3.9a and 3.9b show the time series of the `run_timer_softirq` softirq for AMG and UMT. As we can see from the figures, and as confirmed from previous studies, the `run_timer_softirq` softirq has a long-tail density function.

Table 3.7: Timer interrupt statistics

	freq(ev/sec)	avg(nsec)	max(nsec)	min(nsec)
AMG	100	3,334	29,422	795
IRS	100	6,289	35,734	867
LAMMPS	100	3,763	34,555	1,194
SPHOT	100	1,498	10,204	833
UMT	100	6,451	29,662	982

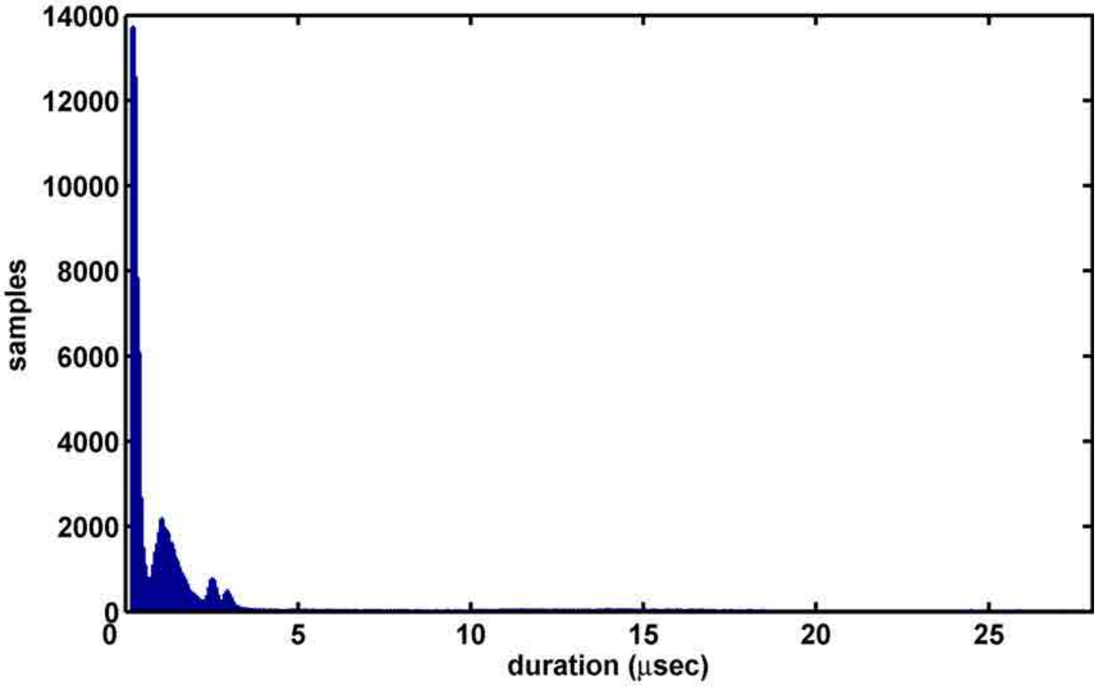
Table 3.8: Softirq *run_timer_softirq* statistics

	freq(ev/sec)	avg(nsec)	max(nsec)	min(nsec)
AMG	100	1,718	49,030	191
IRS	100	3,897	57,663	193
LAMMPS	100	2,242	58,628	256
SPHOT	100	620	32,926	223
UMT	100	3364	87,472	214

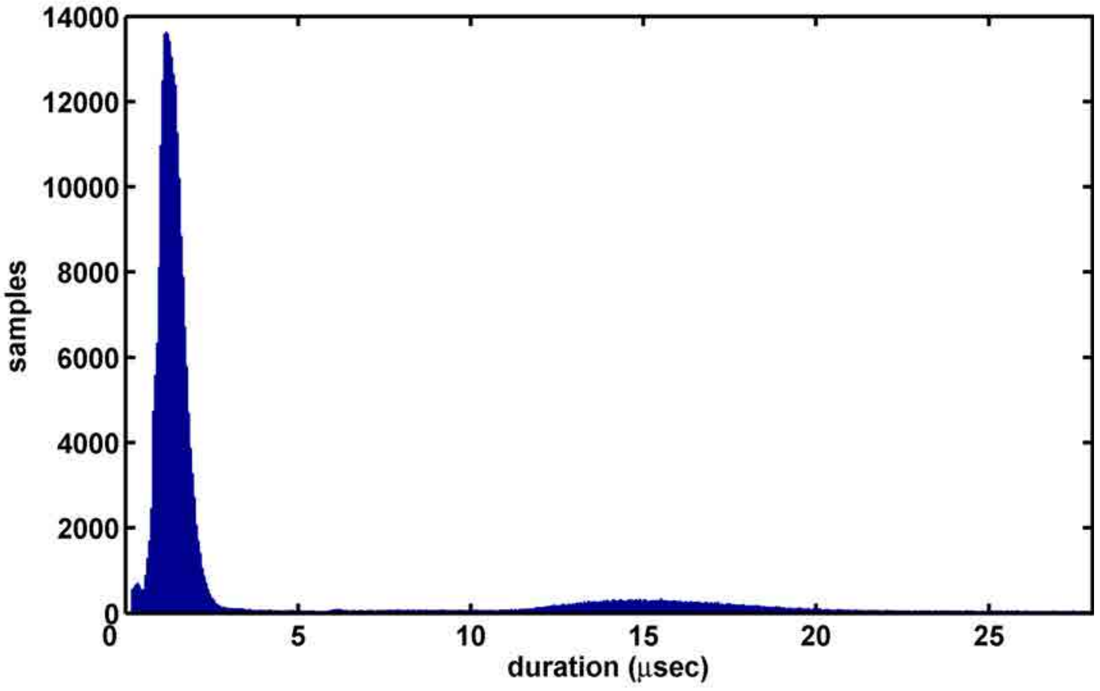
Table 3.7 and 3.8 show statistical data related to the timer interrupt and `run_timer_softirq` softirq. As expected, the frequency of the timer interrupt is at least 100 events/second (100 Hz) for all the applications. Also, the fact that the frequency is not higher means that the applications do not set any other software timer.

3.6 Noise disambiguation

Analyzing OS noise indirectly through micro benchmarks may hide the true causes of a given noise event and lead the developer towards the wrong direction. This section provides two example of using LTTng-noise to disambiguate similar OS activities.



(a) AMG



(b) UMT

Figure 3.9: `run_timer_softirq` time series

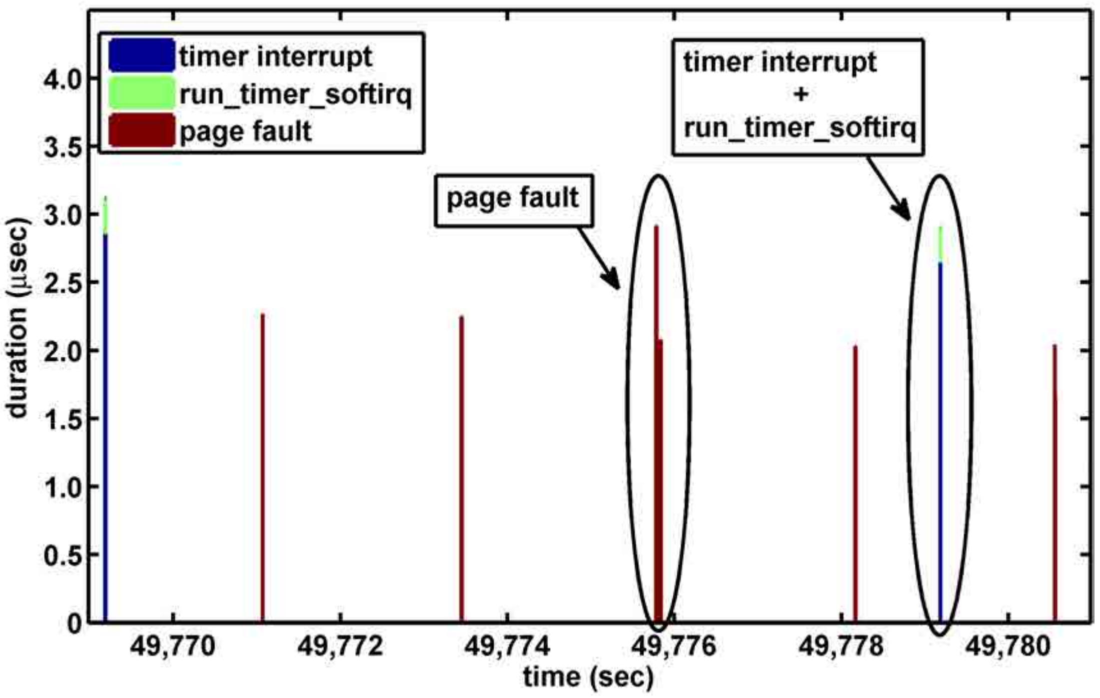


Figure 3.10: AMG - Synthetic OS noise graph

3.6.1 Disambiguation of qualitative similar activities

Figure 3.10 shows a portion of the synthetic OS noise chart for *AMG*. The graphs shows several page faults (red), and two timer interrupts (blue) which, in turn, trigger the `run_timer_softirq` softirq (green). The picture shows that there are two interruptions, a page fault and a timer interrupt (both highlighted in the graph) that have similar duration (2913 *nsec* and 2902 *nsec*, respectively). Indirect measurements through micro benchmarks do not permit to distinguish between these two kernel activities, thus, the developer may think that they are the same activity.

The synthetic OS noise chart provided by LTTng-noise, instead, allows us to disambiguate the two OS interruptions and to clearly identify each activity. In this particular case, the graph shows that the first highlighted interruption is a page fault that takes 2913 *nsec*, while the second interruption is composed by a timer interrupt handler (2648 *nsec*) and `run_timer_softirq` softirq (254 *nsec*), which sum up to a total of 2902 *nsec*.

3.6.2 OS noise composition

Micro benchmarks, such as FTQ, compute the OS noise as missing operations in a given iteration[154]. This means that micro benchmarks are not able to distinguish two unrelated events it they happen in the same iteration.

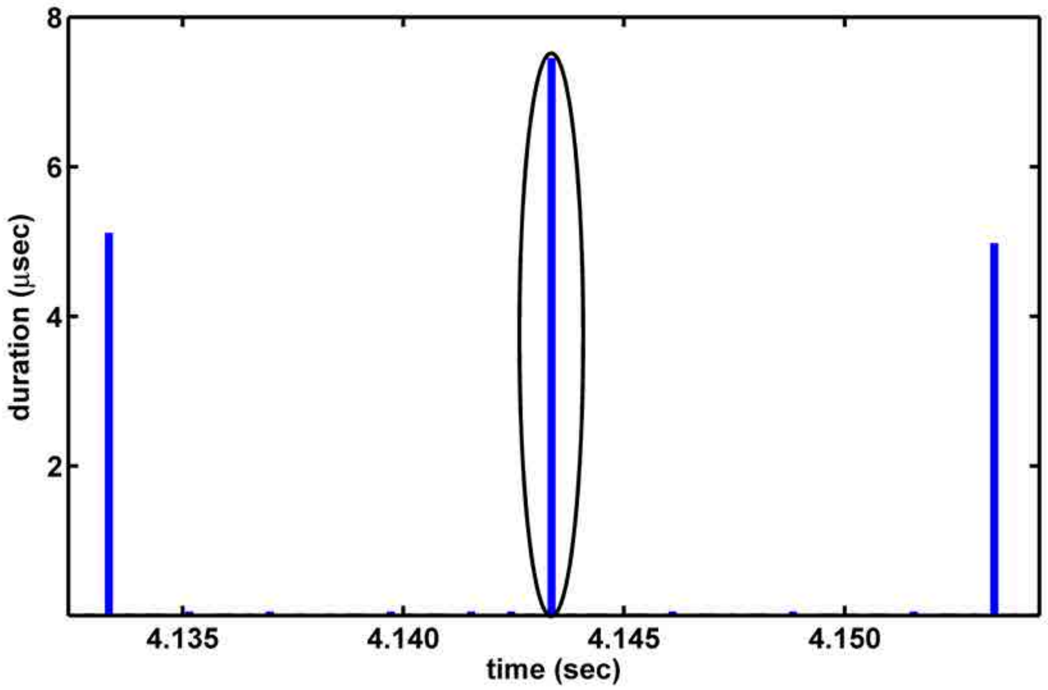
Figure 3.11a shows three iterations of the FTQ micro benchmark. The three spikes are equidistant (which suggests a common periodic activity) but the jitter measured in the first and the third iterations (about 5 μsec) is different from the one measured in the second iteration (7.5 μsec). This different OS noise suggests that the events occurred during the first and the third iterations are different from those occurred during the second iteration. Moreover, the spikes in the first and third iterations are similar to the very frequent ones caused by the timer interrupt while the spike in the third iteration is not similar to anything else. A qualitatively analysis would conclude that FTQ experienced a timer interrupt during the first and the third iteration while "something else" happened during the second iteration, contradicting the hypothesis of equidistant events.

Our analysis shows that, indeed, a timer interrupt also occurred during the second iteration (Figure 3.11b), confirming the first observation of equidistant events. However, right before that timer interrupt, a page fault occurred. In this case, FTQ was not able to distinguish the two events that, indeed, appear as one in its graph. LTTng-noise, instead, precisely shows the two events as separate interruptions, allowing the developer to derive correct conclusions about the nature of the OS noise. This example shows how an internal analysis is more effective than an indirect one and provides information that, otherwise, would be impossible to obtain.

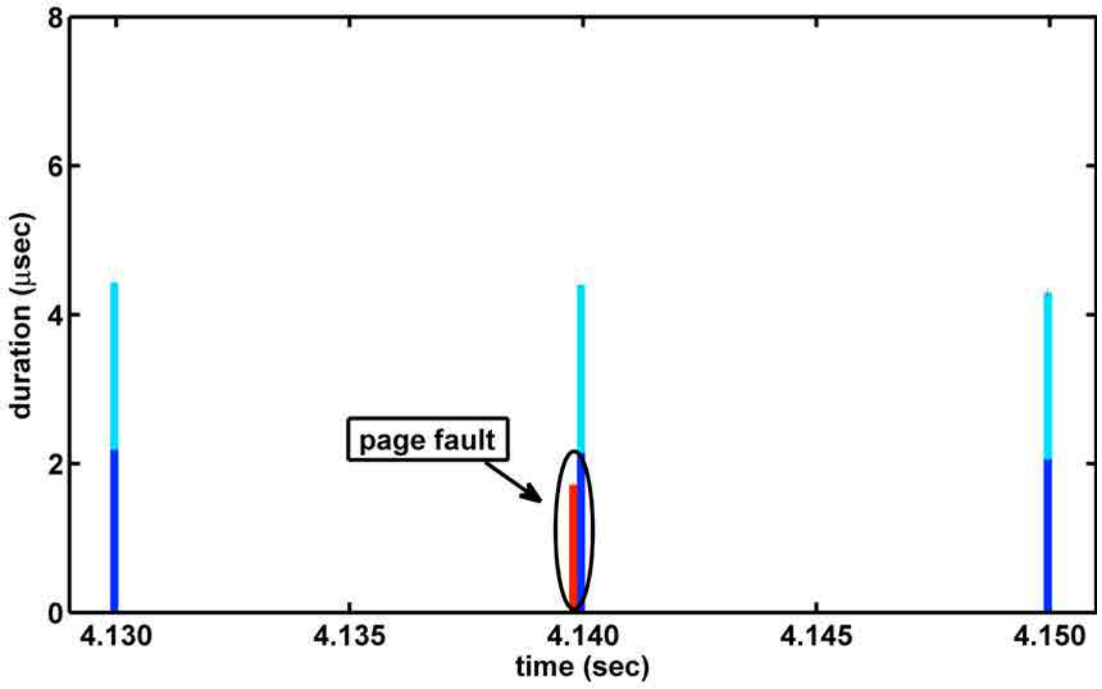
3.6.3 Conclusions

OS noise has been studied extensively and it is a well-known problem in the HPC community. Though previous studies succeeded to provide important insights on OS noise, most of those studies are "qualitative" in that they characterize the overall effect of OS noise on parallel applications but tend not to identify and characterize each kernel noise event. As HPC applications evolve towards more complex programming paradigms that require richer support from the OS, we believe that a quantitative analysis of the kernel noise introduces by such operating systems is most needed.

In this chapter we presented our technique to provide a quantitative descriptive analysis for each kernel noise event. We extended LTTng, a low-overhead, high-resolution kernel tracer, with extra trace points and offline modules to analyze the OS noise. In



(a) OS noise as measured by FTQ



(b) Synthetic OS noise chart

Figure 3.11: Noise disambiguation

particular, we developed a module that generates execution traces suitable for Paraver and a data format that can be used as input for Matlab. We demonstrated that our new technique and toolkit (LTTng-noise) accurately reflect the noise measured by other known techniques (such as FTQ). In addition to previous insights, our new technique allows quantifying and providing details for what contributed to the noise for each interruption.

Using LTTng-noise, we analyzed the OS noise introduced by the OS on LLNL Sequoia applications and showed that 1) each application experiences different jitter (both in terms of overhead and composition), 2) page faults may have even larger impact than timer interrupts (both in terms of frequency and duration), and 3) some activities have larger time distributions that may lead to load imbalance at scale for particular applications. Moreover, we are able to identify and quantify each single source of OS noise. This capability is very useful to analyze current and future applications and systems.

Finally, we showed two case studies where we used our technique to disambiguate kernel noise events. This noise disambiguation would not be possible without the detailed information provided by LTTng-noise.

We plan to use LTTng-noise do to deeper analysis of current and future parallel applications and to quantify how our findings affect the scalability of those applications on large machines with hundreds of thousands of cores.

Chapter 4

Light-weight Operating Systems

This part of the thesis addresses the problem of operating system scalability. In the previous chapter we examined the scalability issues of general purpose operating systems, while in this chapter we focus on the scalability of lightweight OS. IBM CNK is a lightweight kernel developed for the BlueGene systems that enables high scalability with low overhead. As explained in the introduction, the main limitation of a lightweight kernel, with respect to a general purpose OS, is the limitation in terms of features. CNK for instance, lacks the support for dynamic memory mapping (i.e. , virtual to physical memory mapping is determined at loading time and never changes). In this chapter, we perform a quantitative study of the cost of adding dynamic memory mapping to CNK in term of application performance and scalability.

4.1 Summary

In order to achieve 10^{18} FLOPS with a limited power budget (25MW) [24], exascale computing will induce novel technologies and fundamental changes in the software stack [58]. Exascale systems are expected to feature O(1000) more cores than current systems (up to 10^8 - 10^9 cores) but the growth in volume of memory is not expected to match that of processor cores. Exascale systems are, thus, envisioned to have orders of magnitude less memory per core than current petascale systems, in terms of both total memory per node and cache memory per core. To reduce the complexity posed by the high concurrency level (order of billions of threads) and limited available per-core memory, programmers are already exploring novel programming models and algorithms. Modern parallel applications are becoming more complex: UMT from the LLNL Sequoia benchmarks [110], Climate code [100], and UQ [23] require dynamic library and script language support, as well as a richer support from the OS and the system software. Moreover, there is a growing interest in richer shared-memory programming models, such as OpenMP [134], PGAS (e.g., UPC[60, 161] and GA[128]), Charm++[96], and Transactional Memory [147, 87, 83], that have the potentiality to simplify parallel programming and to enable users to extract higher level of parallelism. Finally, dynamic libraries, python scripts, dynamic memory allocation, checkpoint-restart systems, tracers to collect performance and debugging information, and virtualization [104], are seeing wider use in HPC systems. Current trends indicate that large petascale to exascale systems will desire a richer system software ecosystem with support for all those technologies and more complex programming models. In this scenario, memory management becomes of predominant importance and, therefore, one of the operating system components that might undergo a fundamental re-design.

Current petascale systems, however, have taken opposite approaches, especially for what concerns memory management. Petascale supercomputers run either general purpose operating systems (OS), such as Linux, adapted to run high performance computing (HPC) applications, or lightweight kernels (LWKs), such as CNK [72] or Catamount [140], specifically designed to run scientific code. Both solutions provide advantages and disadvantages: general purpose OSes provide high flexibility and a large variety of runtime services that virtually allow running any kind of application and make them suitable to run future HPC applications. At the same time, general purpose OSes also introduce considerable noise, which has been identified as one of the source of poor scalability on large systems [68, 75, 137]. Lightweight kernels are explicitly designed to efficiently run parallel applications (more specifically, MPI and/or OpenMP applications) but often require internal modifications when adding support for a new service [119]. Constantly adding new services and flexibility might increase the level of system noise.

We present a study on the impact of the memory management sub-system on modern and future parallel HPC applications, with particular focus on the effect of the translation look-aside buffer (TLB) misses at scale, to understand whether a more flexible approach than current LWK's solution is practical from the performance point of view. Although adding support for TLB misses necessarily introduces some runtime overhead, a flexible TLB management would allow the development of the richer ecosystem required by exascale applications.

The impact of TLB misses on real HPC applications has been studied in the past but only qualitatively [148], mainly because of the difficulties in comparing different solutions using the same OS. In this study, we provide an apples-to-apples comparison and analyze different TLB management approaches — dynamic memory mapping, static memory mapping with replaceable TLB entries, and static memory mapping with fixed TLB entries (no TLB misses) — on a real BG/P system with up to 1024 nodes (4096 cores). To this extent we modified CNK, a noiseless OS designed for Blue-Gene systems, by adding support for TLB misses. CNK was not originally designed to take TLB misses, thus we implemented the required functionalities, including exception handler and replacement policy, to resolve virtual-to-physical mappings. We also implemented a lightweight tracing mechanism to collect per-TLB miss statistics without introducing excessive overhead.

We, then, perform a sensitivity analysis study varying the number of concurrent nodes, the size of the pages (from 4KB to 16MB) and the overhead introduced by the TLB miss handler. Finally, in order to simulate a more complex memory management sub-system, such as one with dynamic memory mapping, and evaluate the effect of TLB misses at scale while avoiding the noise of other OS components, we inject delays with random and constant signatures each time a TLB miss occurs.

We consider that a performance degradation of 2% would be justified by the higher level of thread level parallelism that a richer system software ecosystem could help achieve. We analyze the TLB pressure and the overall performance degradation of several applications from the LLNL Sequoia [110] and ANL benchmark suites. Our experimental results show that, for several applications (including AMG, IRS, LAMMPS, and GTC), the impact of TLB misses is below 2% with 1MB-pages even when injecting a large artificial TLB miss noise signature. For UMT, a more complex application, using 1MB-pages still introduces considerable overhead, especially when injecting extra TLB miss overhead, while 16MB-pages do not introduce more than 2% performance degradation. This conclusion opens the possibility of implementing richer memory management services at OS level to satisfy the requirements of future applications and users.

4.2 Experimental environment

Table 4.1 show the experimental environment used in this chapter. The table includes the hardware system, the operating system, the communication library, and the benchmarks used for the experiments.

CPU	PowerPC 450 at 850 MHz (4 cores)
Memory	4 Gigabyte of DDR2 RAM
Network	3D toroidal custom network
System size	up to 1024 nodes (4096 cores)
Operating System	CNK
Communication Library	MPI (IBM)
Benchmarks	Sequoia benchmarks from LLNL (AMG, IRS, LAMMPS, UMT) and GTC from ANL

Table 4.1: Experimental environment - IBM BlueGene P system

4.3 Related Work

The importance of TLB handling for system performance has long been studied in the past [160, 141]. Anderson et al. [14] showed that the TLB miss handler is one of the most frequently executed kernel function and that, in corner cases, handling TLB misses can take up to 40% of the execution time [90]. A following work [98] reports that data TLB handling can take up to 10% of the runtime of SPECCPU 2000 benchmarks. A recent study [28] characterized the TLB behaviour of applications running on chip multiprocessors, confirming that TLB data and instruction misses can hinder system performance significantly, and that between 30% to 90% of TLB misses are either predictable or redundant among multiple threads. McCurdy, Cox and Vetter [116] also studied the TLB profile of floating point portion of SPECCPU benchmarks and the HPC Challenge suite. The authors found that the TLB miss profiles of these benchmarks can be significantly different from production applications. Moreover, this work shows that a wrong page size choice may result in a performance degradation of up to 50% for production applications.

Several studies in the literature propose techniques to reduce the impact of TLB misses. Bhattacharjee and Martonosi [29] propose an inter-core cooperative TLB for chip multiprocessors that uses prefetching to exploit commonalities between TLB miss patterns among cores. The same authors also propose the use of a shared last-level TLB for chip multiprocessors [27], exploiting the fact that multiple threads experience TLB misses on identical address translation entries.

While all the mentioned works address the impact of TLB miss on application and system performance, this work is the first, to the best of our knowledge, that isolates and characterize the effects of TLB misses on large clusters and real HPC applications.

4.4 Memory management in general purpose OS and lightweight kernels

Memory management is one of the key points in which general purpose and lightweight kernels have taken opposite approaches. Linux-like operating systems support memory protection among different concurrent applications, dynamic memory allocation and remapping, virtualization, etc. Virtual-to-physical mapping can be dynamically modified during the application execution: to each process is assigned a set of page tables that store the current translation from virtual-to-physical mapping of any page. These features provide high flexibility to the programmers but also introduce overhead and performance variability: since virtual-to-physical mappings can change at run-time, the OS may need to retrieve the page table that contains the physical address corresponding to a given virtual address. On the other hand, lightweight kernels often provide only a reduced set of the previous services and, in particular, may use static virtual-to-physical mapping, i.e., the virtual-to-physical mapping of an application is determined at the beginning of the application and does not change throughout its execution.

Translation look-aside buffers are commonly used to reduce virtual address translation latency: a small set of page table entries (from ten to a few thousands) is cached into the TLB and can be quickly accessed to retrieve the physical address associated with the virtual address of the desired page. During a memory access (both for data or instruction), if the virtual-to-physical mapping of the desired page is not in the TLB (TLB *miss*), the system needs to retrieve the page table entry that maps the virtual address to the corresponding physical one, then allocate the page table entry into the TLB and finally issue again the same memory operation (that this time will hit in the TLB). This process can be done by the hardware (hardware-assisted TLB), transparently to the software, or explicitly by the OS (software-managed TLB). In both cases, a TLB miss introduces run time overhead that, as well as other system activities, may limit scalability on large machines [148].

In this work we examine three different approaches to map virtual addresses to physical ones:

- **Fixed TLBs**: With this approach virtual-to-physical mappings are statically determined at the beginning of the application and do not change throughout the execution (static memory mapping). The mappings are stored in the TLB entries and are never replaced (i.e., there are no TLB misses).

- **Replaceable TLBs**: As in the previous case, virtual-to-physical mappings are statically determined at the beginning of the application and do not change. However, to the contrary of the previous approach, TLB misses may occur, i.e., the entries stored in the TLB can be replaced at runtime.
- **Dynamic mapping**: This approach is the most flexible: virtual-to-physical mappings can change at runtime (dynamic memory mapping). TLB entries store a subset of the current virtual-to-physical mappings and can be replaced at runtime.

With dynamic memory mappings, the virtual-to-physical translations of a process can be modified at runtime by the OS. This approach allows memory-on-demand (the virtual-to-physical mapping is determined at the moment a process first addresses a new page), copy-on-write (the virtual-to-physical mapping of a page is modified when the first write to a page shared by parent and child process occurs), swapping (the physical address of a page temporary swapped on disk is determined after the page is loaded back to memory), virtualization, etc. All these techniques are commonly employed in general purpose operating systems, such as Linux: they usually provide considerable advantages for desktop or data centers systems but also introduce time variability and larger overheads. Several LWKs (e.g., CNK) have, instead, taken the approach of statically mapping page table entries to TLB entries at the beginning of the execution of an application and avoid TLB misses altogether (fixed TLBs). Replaceable TLBs are less restrictive than fixed TLBs in that, even if the virtual-to-physical translation mapping of an application does not change throughout the execution, the mapping may contain more entries than provided by the hardware, hence TLB misses occur to allow more complex mapping layouts than the one offered by fixed TLB mappings. With static memory mapping (fixed TLBs and replaceable TLBs) pages cannot be remapped at runtime, hence, the dynamic techniques described above are not easy to implement. On the other hand, the OS design is simpler and there is minimal runtime overhead.

In order to provide the services required by future parallel applications and satisfy, at the same time, the hardware/technology/cost constraints, operating systems may undergo a substantial changes in the exascale era.

For example, according to a common knowledge, TLB misses introduce a considerable amount of OS noise and limit scalability on large clusters. Common solutions to this problem are static memory mapping (CNK, Catamount) or the use of large pages (HugeTLB). TLB misses are introduced because of the "dynamic" behavior of the OS. This dynamic behavior is caused by OS services (memory protection among different applications, page swapping, dynamic memory, virtualization, etc.) or optimization (COW, memory on demand, etc.). However, allowing TLB misses does not necessarily imply that all those services/optimization have to be implemented. We can allow TLB misses but not memory on demand or COW, for example. We cannot do the other way around, though (or it would be difficult). Which service/optimization we should implement requires a different study.

The new generation of Exascale Supercomputers poses several challenges for the OS design. Effective design of a reliable, performing, memory management is one of the biggest challenges, and the use of a static or dynamic virtual address translation mechanism is a key decision. On one hand, a static translation mechanism is safest choice in term of performance, on the other hand it considerably worsen the memory defragmentation. In order to choose between this two designs, we need to fully understand the impact of a dynamic translation mechanism with respect to the static one. Our goal is to provide enough evidences to support or reject the hypothesis the TLB misses performance impact is relevant for a realistic HPC workload.

4.5 Methodology

In order to understand the effect of TLB misses on scientific applications previous work compared [148] an OS with fixed TLB entries (e.g., CNK) with one that uses dynamic TLB mapping (e.g., Linux). Indeed the performance difference between a general purpose OS and a LWK specifically designed to run HPC applications may be considerable.

Figure 4.1 shows the execution time of AMG solving phase when running with CNK and with a non-optimized version of Linux on BG/P and varying the number of MPI processes (cores) from 128 to 2048. The Figure shows that 1) the overhead of Linux over CNK for 128 node is already large (3.62x), and that 2) the overhead increases with the number of cores (up to 6.86x), which indicates limited scalability. The problem with this approach is that Linux introduces OS noise that is not necessarily caused by TLB misses (such as process preemption, CPU migration, or interrupts), hence it is difficult to isolate the TLB noise or even understand if the overhead introduced by TLB misses is a major problem.

Our approach, instead, consists of modifying CNK, a noiseless OS designed for

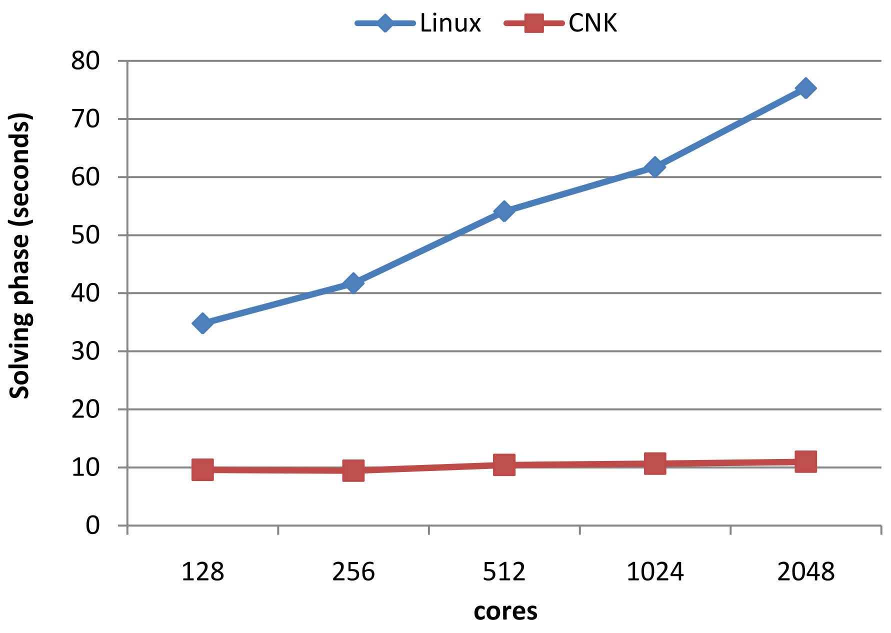


Figure 4.1: Performance of AMG in weak scaling mode with a non-optimized version of Linux and with CNK on BG/P. This figure shows the effects of Linux system noise on AMG at scale: compared to CNK, Linux shows lower performance (3.62x with 128 cores) and limited scalability.

BlueGene/P systems, by introducing support for TLB misses and then comparing the original CNK with our modified version (CNK-TLB). With this approach the effect of TLB misses is isolated from the rest of the OS noise introduced by a full-fledged OS such as Linux. For this study, we used a BG/P cluster: The value of using a BG/P over other architectures and/or simulators is two-fold. First, we can perform experiments at a much larger scale than in a simulated environment, which provides more representative results for the supercomputing environment. Second, BG/P cores (PowerPC 450) do not feature an hardware page-lookup circuit: TLB misses are handled by software, thus we can explore different TLB miss strategies by specifically designing the proper TLB miss handler.

4.5.1 Adding support for TLB misses to CNK

The overhead of translating a virtual address into the corresponding physical one in our system depends on several components difficult to isolate, such as the TLB miss exception handler latency (including kernel/user context switches), the virtual-to-physical translation algorithm, and the TLB replacement policy. To isolate the overhead of TLB miss exceptions from the other components (virtual-to-physical translation algorithm and TLB replacement policy), we implement a low-overhead, range-checking, roundrobin TLB miss handler. PowerPC 450 has 64 TLB entries per core, out of which 11 are statically fixed and reserved to map the CNK address space; all the other entries store application virtual-to-physical mappings.

CNK was not designed to support TLB misses, hence virtual-to-physical mappings are determined at loading time, assigning fixed segments to the application. A TLB miss normally would raise an exception reporting an error and aborting the running application. CNK uses four memory segments per process, namely text, data, heap/stack and shared. The position of each memory segment in the physical address space is determined and saved into special registers when the binary is loaded. There is one private data and heap/stack segment for each process, while the text and the shared data segments are common to all processes in a node. A schema of virtual-to-physical memory mapping for two running processes (SMP running mode) is depicted in Figure 4.2. In order to translate virtual addresses we implemented an exception handler that uses special registers to memorize the beginning of each memory segment and perform a translation based on the range of the virtual address. Furthermore, the standard CNK segments are aligned to 1MB, and using larger pages would cause the segments not to be aligned with the pages. To solve this issue, when handling 16MB-pages, our TLB handler uses a minimum (< 16) number of 1MB-pages to fill the gap and then uses 16MB-pages for the rest of the segment.

Every time a TLB miss occurs, the TLB miss handler translates the virtual address as an offset from the proper segment in the physical address space. Since segments positions are memorized into special registers, this approach does not require access to memory to retrieve the virtual-to-physical mapping, thus the effect of data cache misses is reduced. Nonetheless, memory accesses are still required for the TLB exception handler context switch.

4.5.2 Tracing TLB misses

Efficient and precise measurement of TLB misses requires considerable memory space, which may introduce additional data cache misses and, therefore, invalidate our results. To guarantee low overhead, we record the start and end time of the TLB miss handler in memory and then dump the information to disk at the end of the application.[1] Considering the size of PowerPC 450 timestamp (8 bytes) and the number of TLB misses per second (up to 10^6 per second with 4KB pages), the size of data memorized in the buffer can reach more than 10MB per second. This amount of data

[1]Note that the measured interval does not include the time to save and restore the first five registers used in the TLB handler (context switch).

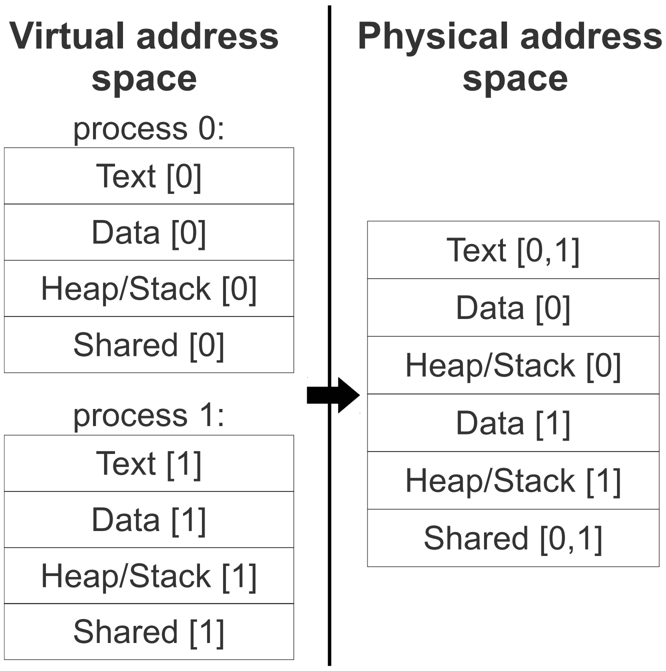


Figure 4.2: Example of virtual-to-physical memory mapping for two MPI processes (SMP mode) in CNK.

would definitely affect the application memory footprint, providing biased results. We, thus, implemented sampling techniques, at process and TLB event levels, to reduce the amount of data while maintaining representativeness. A set of MPI process is selected at the beginning of the applications as the ones that will record TLB miss events. With 64KB pages the sampling select 1 TLB miss every 64 to be recorded, while with 4K pages 1 every 256. To validate the sampling frequency we compare the data obtained with and without the sampling mechanism. We compare the two measured TLB miss samples performing a two-sample Kolmogorov-Smirnov test with a 95% confidence interval. Table 4.2 shows the validation for the sampling with 64KB pages, reporting the arithmetic mean, the standard deviation and the result of the statistical test for the TLB duration distributions.

Notice that this approach does not reduce accuracy because scientific applications are single process/multiple data (SPMD), i.e., each process performs the same operation on a different input set, and because HPC applications are usually iterative, i.e., it is possible to extract applications' characteristics, such as the TLB miss pattern, from a few iterations.

Figure 4.3 shows a partial execution trace of AMG in which we recorded all TLB

Table 4.2: TLB duration distribution: sampling with 64KB pages

		mean		**std dev**		**test**
Apps	**Cores**	**full**	**sampled**	**full**	**sampled**	
amg	128	72.63	72.53	16.77	16.37	passed
irs	125	76.43	75.91	20.59	18.09	passed
lammps	128	69.46	69.50	5.36	5.89	passed
umt	128	68.88	68.86	2.48	2.40	passed
gtc	128	71.52	71.55	13.11	12.87	passed

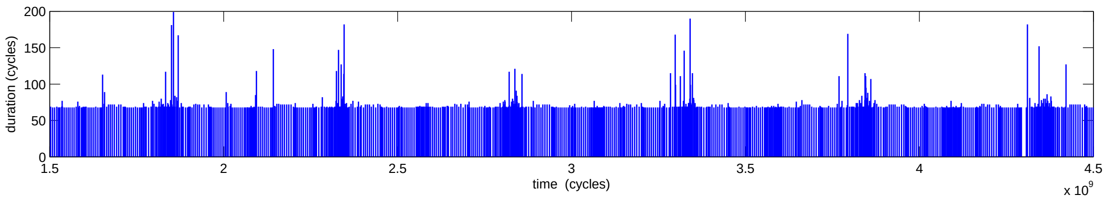


Figure 4.3: TLB miss trace of 3.52 seconds of AMG execution.

misses: y-axis is the duration of the TLB miss and x-axes represent the timeline (3.52 seconds in total). As we can see from the trace, AMG presents a iterative structure, thus sampling either one complete iteration or enough events throughout the execution does not lead to measurable inaccuracy.

We measure the overhead of our tracing system by comparing the instrumented version of CNK-TLB with the non-instrumented version and found it negligible (about 1%, depending on the application and in the same order of the performance variation across iterations) but, to provide more accurate results when measuring the execution time overhead, we compare the original CNK kernel to the non-instrumented version of CNK-TLB and we use the tracing version only when reporting per-TLB miss statistics.

4.5.3 TLB noise injection

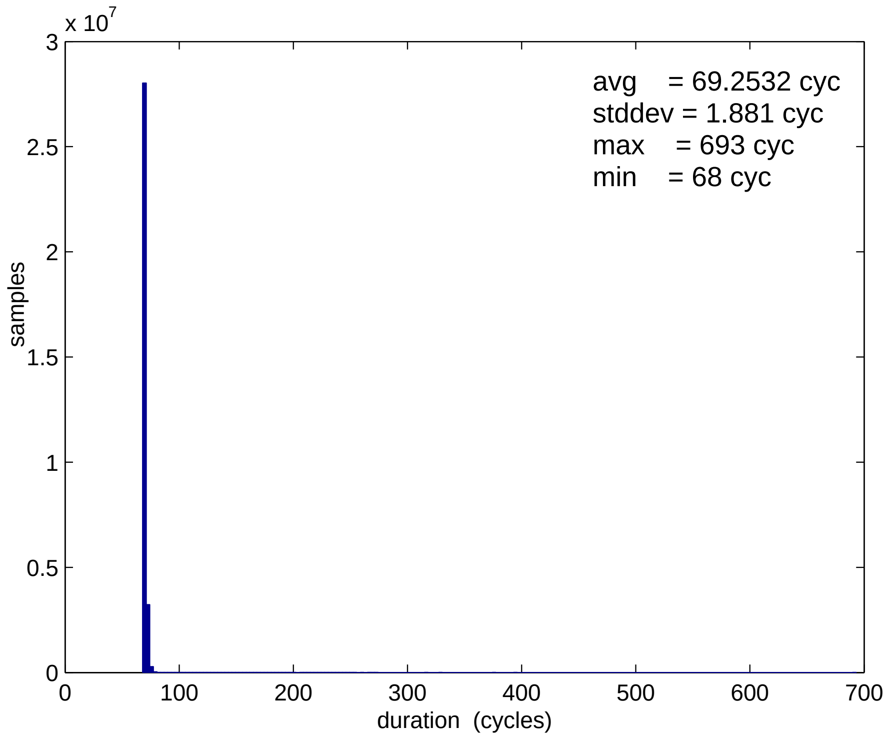
CNK-TLB uses a simple range-checking TLB handler (described in Section 4.5.1) that introduces minimal overhead: The range-checking TLB handler implemented in CNK-TLB shows a very tight time distribution centered around 70 cycles. In reality, however, operating systems use different techniques to resolve virtual-to-physical mappings, such as paging or segmentation [36]. These techniques require looking up for the page table or the segment that contains the virtual-to-physical mapping when a TLB miss occurs. Besides the overhead of computing the address of the correct page table, paging may require several accesses to memory and, eventually, the allocation of new page tables (in case of the first access to a page) before translating the virtual address.

This results in a duration distribution of the TLB miss handler that is not as tight as the one obtained with CNK-TLB. CNK-TLB uses a low-overhead, range-checking TLB miss handler while Linux TLB miss handler is affected by several components (page table lookup, cache misses, page table allocation, etc.) that produce a wider time distribution. Figure 4.4 shows the TLB miss handler execution time distribution for CNK-TLB (Figure 4.4a) and for Linux (Figure 4.4b) when running UMT with 1024 MPI processes. As we can see from the figures, the Linux distribution is not as tight as the CNK one and presents several peaks, each of which indicates a different event.

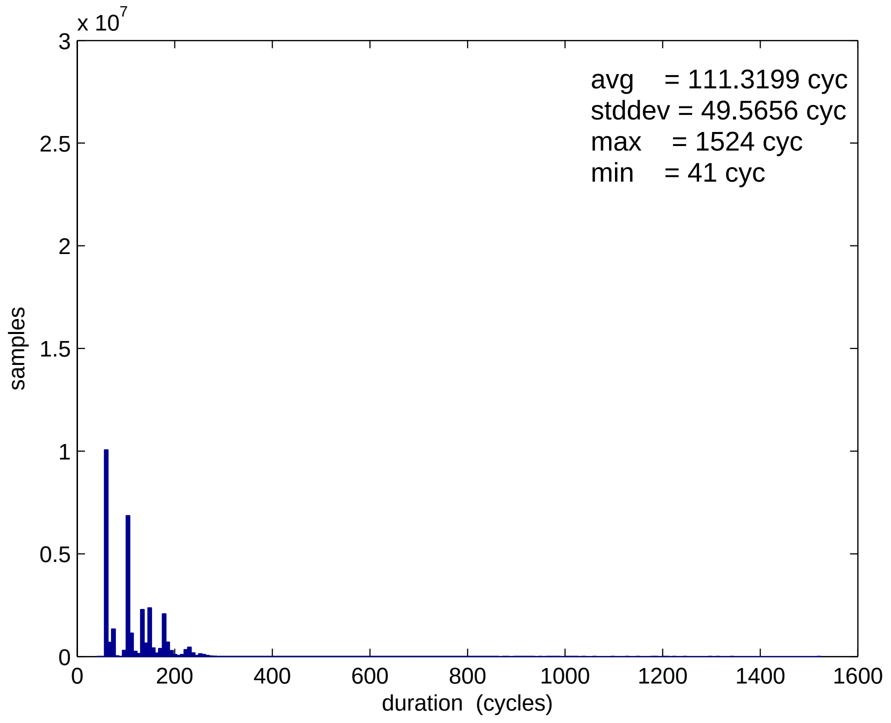
While the implementation of paging in CNK is out of the scope of this work, we want to more accurately evaluate the overhead introduced by TLB misses on an HPC application running on a system with paging. To this extent we use *noise injection* [68], a technique that consists of injecting artificial delays with different signatures to simulate the noise introduced by a real OS component (in this case, the memory sub-system). We inject two different signatures in CNK-TLB: a constant signature and a uniformly distributed random signature. In both cases we vary the maximum delay injected on each TLB miss to understand what is the maximum per-TLB miss overhead that guarantees overall performance slowdown below 2%. At each TLB miss, the TLB handler selects the delay to add, either the constant delay (constant signature) or one random value from the uniform distribution (random signature), and executes a loop to delay the TLB miss handler. To minimize the overhead of selecting the random delay, we load the uniform random distribution at application start time in kernel memory.

4.6 Experimental results

In this section we analyze the performance impact of TLB misses overhead from the LLNL Sequoia (AMG, IRS, LAMMPS, and UMT) and from ANL (GTC) benchmark suites. In order to isolate the effects of TLB miss overhead, we compare the solving phase execution time of the applications running on CNK, a noiseless OS that does not take TLB misses, to our modified version of the same OS (CNK-TLB), which is capable of handling TLB misses. As described in the previous section, our TLB miss handler uses a low-overhead range-checking approach that does not need to look for page tables in memory to resolve virtual-to-physical mappings. Although this approach is a simplified version of a normal system with memory paging, it provides the minimum overhead caused by a TLB miss, i.e, the overhead of going to kernel mode and back, the execution of the TLB miss handler, the L1 and L2 instruction cache misses and the



(a) CNK-TLB



(b) Linux

Figure 4.4: TLB miss handler execution time distribution for CNK-TLB and Linux.

overhead of flushing the core pipeline before entering kernel mode and upon resuming the execution of the application. Section 4.6.3 analyzes the effect of page table lookup through TLB noise injection.

4.6.1 TLB pressure

Table 4.3 shows the TLB pressure that each application poses on the TLB cache, i.e., the average number of TLB misses/second per core in the system when varying the number of concurrent MPI processes (cores) and the page sizes. Each value in Table 4.3 is the average of 4 runs)[1]

We report the TLB miss frequency rather than the absolute number because each application runs for a different amount of time, thus the absolute number of TLB misses alone is not a valid indicator of application's TLB pressure. Finally, here, and in the rest of the chapter, we focus on the solving phase of each application, i.e., we do not report TLB misses for the initialization and finalization phases.

The applications tested in our experiments present different characteristics, such as data locality and memory and communication patterns, that have an impact on the TLB pressure. The results is that each application poses a different pressure on the TLB and the memory sub-system. For example, with 1MB pages and 1024 cores, LAMMPS shows 0.64 TLB misses/second while AMG and UMT show 658.69 and 42,957.35 TLB misses/second, respectively, which indicates that UMT poses higher TLB pressure than the other applications. Also, please not that results for UMT are up to 2048 cores, because UMT uses only half of the cores in each node. We expect UMT to suffer higher slowdown on CNK-TLB than the other benchmarks, especially with small pages, On the other hand, the TLB pressure reduces with the page size (the larger the page, the lower the number of TLB misses/second) across all the applications.

Table 4.3 shows that the impact of the page size on the number of TLB misses/second per core is very large. When the page size increase from 64KB to 1MB (16x), the number of TLB misses/second may decrease by up to five orders of magnitude (LAMMPS).

Finally, Table 4.3 reports that the number of TLB misses/second is roughly constant at scale (except for IRS). This is somehow expected because we run our experiments in

[1]In our experiments with CNK-TLB using 4KB-pages cause several applications not to complete (GTC, UMT) or to generate MPI errors (IRS). We omit these results since, as we will see in the next sections, 4KB-pages do not play a determinant role in our conclusions. Also note that for IRS the data set distribution requires a number of processes that is not a power of 2.

[1]Many application did not complete with 4K pages due to the high overhead introduced by TLB misses

Table 4.3: Number of TLB misses/second per core during the solving phase for LLNL and ANL applications using CNK-TLB. The different number of cores for IRS is due to the design constraint of having a number of parallel processes that is a power of three.

		Page size			
Apps	**Cores**	**4K**	**64K**	**1M**	**16M**
amg	128	202,718.24	16,580.67	565.95	0.00
amg	256	217,190.96	18,089.83	685.78	0.00
amg	512	229,597.23	17,732.09	622.96	0.00
amg	1024	232,221.82	18,433.50	658.69	0.00
amg	2048	243,566.54	18,460.25	681.54	0.00
amg	4096	NA	18,834.16	724.70	0.00
irs	125	NA[1]	23,104.27	0.88	0.11
irs	216	NA	24,273.60	0.75	0.09
irs	512	NA	25,035.51	0.59	0.00
irs	1000	NA	25,133.25	0.44	0.05
irs	1728	NA	29,959.76	0.79	0.04
irs	4096	NA	38,506.07	0.00	0.03
lammps	128	1,017,630.05	34,888.56	0.66	0.15
lammps	256	1,061,163.23	37,492.60	0.67	0.15
lammps	512	1,021,721.25	33,275.87	0.66	0.15
lammps	1024	1,085,015.14	33,788.92	0.64	0.15
lammps	2048	1,040,772.45	32,341.94	0.66	0.15
lammps	4096	NA	37,699.52	0.63	0.17
umt	128	NA	260,916.79	50,463.81	1.26
umt	256	NA	265,431.37	45,548.68	1.37
umt	512	NA	269,942.34	44,975.80	1.46
umt	1024	NA	275,927.00	42,957.35	1.44
umt	2048	NA	274,559.97	39,277.12	1.77
gtc	128	NA	5,812.62	108.51	0.00
gtc	256	NA	6,413.69	251.87	0.00
gtc	512	NA	6,386.41	251.27	0.00
gtc	1024	NA	6,537.21	250.00	0.00
gtc	2048	NA	6,501.46	248.61	0.00
gtc	4096	NA	6,462.73	253.83	0.00

weak scaling mode, thus the amount of work (and the memory footprint) per process is supposed to be constant. There is a small percentual variation caused by the not-perfect weak scalability of the applications.

4.6.2 Analysis of TLB overhead at scale

Figure 4.5 depicts the performance overhead experienced by the tested applications when varying the number of MPI processes (cores) and the page size.

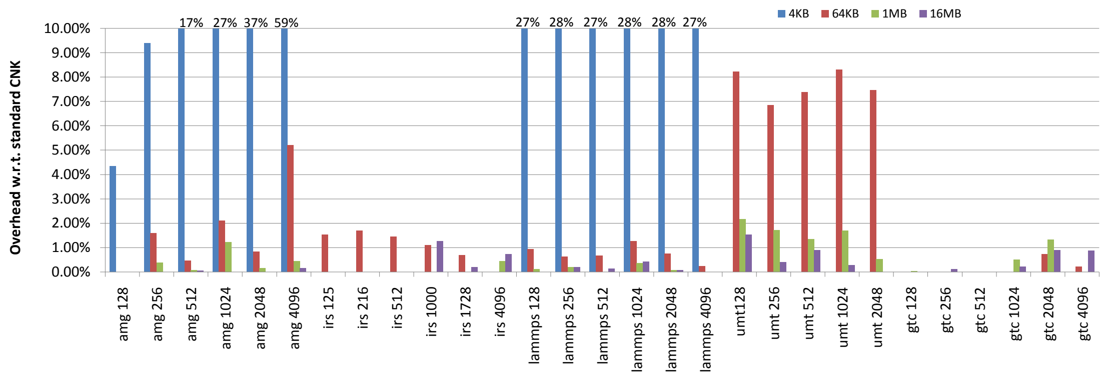


Figure 4.5: TLB misses overhead with varying page sizes and concurrent cores. This graph shows that the overall overhead is below 2% for 16MB-pages for all the applications while 1MB-pages only guarantee overhead below 2% for some application.

We compute this execution time overhead as the ratio between the execution time of the original CNK (fixed TLBs) and our modified version CNK-TLB (replaceable TLBs) without instrumentation. The execution time does not include the data initialization phase, i.e., we only instrument the so-called solving phase. The graph shows that with 4KB-pages the overhead introduced by TLB misses is excessively high for both AMG and LAMMPS in all tested configurations (128-4096 cores), similarly to what reported by Shmueli et al. [148] (although with different applications). With 64KB-pages, the overall slowdown considerably reduces for most of the applications: only UMT and AMG show overhead larger than 2% (8.2% and 5.2%, respectively). We consider 2% as a threshold above which the overhead is too high to take the replaceable TLBs design into consideration for future HPC systems. This threshold may seem too strict but we should remark that, in the experiments in Figure 4.5, CNK-TLB uses a range-checking TLB handler and that the actual overhead of page table lookup is larger (see Section 4.6.3). With 1MB and 16MB pages, the TLB noise reduces below 2% also for UMT and AMG (except UMT with 128 cores), with 16MB pages introducing less than 1% slowdown for most of the applications. This results confirms that the performance degradation induced by TLB misses is contained and sustainable for HPC applications if the TLB pressure is not excessively high (1MB and 16MB cases).

Figure 4.5 also shows that the performance degradation suffered by each application varies according to the application's characteristics. For example, although LAMMPS shows a higher number of TLB misses/second (32K-37.5K) than IRS (23K-38K) with 64KB pages, the overall performance degradation is slightly higher for IRS, which means that each TLB miss introduces larger delays on IRS than on LAMMPS. This happens because the impact of TLB misses on the overall application performance depends on several factors: 1) the TLB pressure (reported in Table 4.3), 2) the TLB handler execution time, and 3) the criticality of the TLB miss. The number of TLB miss and the duration of the TLB miss handler determine the *direct overhead* each process suffers because of TLB misses. This overhead, however, may not directly impact the overall application performance.

As we mentioned in the previous section, each application presents different intrinsic characteristics, such as the communication pattern or the synchronization structure. The overhead of a TLB miss may impact differently each application, depending on the current activity of the process experiencing the TLB miss or whether the process was on the application critical path (*critical miss*) or not (*non-critical miss*). For example, if the process that experiences a TLB miss is the last one to reach a barrier, the TLB miss overhead will directly impact on the overall application performance. Vice versa, if the process is not the last one to reach the barrier, the overhead of the TLB miss will be completely absorbed and will not slowdown the application. This phenomenon is not just related to TLB misses but applies to system software noise in general [137, 75, 68]. For example, a timer interrupt issued while the current process is waiting for an incoming packet does not introduce any delay on the overall applications. Timer interrupts that delay computing phases of processes in the critical path, instead, directly impact on overall performance.

Applications' characteristics and size of the page also influence the duration of each TLB miss handler. In fact, even our range-checking TLB miss handler may suffer different instruction and data cache misses, depending on how frequently TLB misses occur.

Figure 4.6 shows that the execution time of a TLB handler varies from application to application and that, in general, increases with the size of the page. For example, the average TLB miss handler execution time for IRS running with 216 cores is 75 cycles with 64KB pages, 180 cycles with 1MB pages and 325 cycles with 16MB pages. As shown in Table 4.3, the TLB pressure reduces with the size of the page, thus large pages induce fewer TLB misses/second. This means that there is a higher probability of instruction/data cache misses with larger pages than with small ones.

Figure 4.7 shows that, indeed, there is an empirical relationship between the average TLB handler duration and the number of TLB misses/second: the higher the number

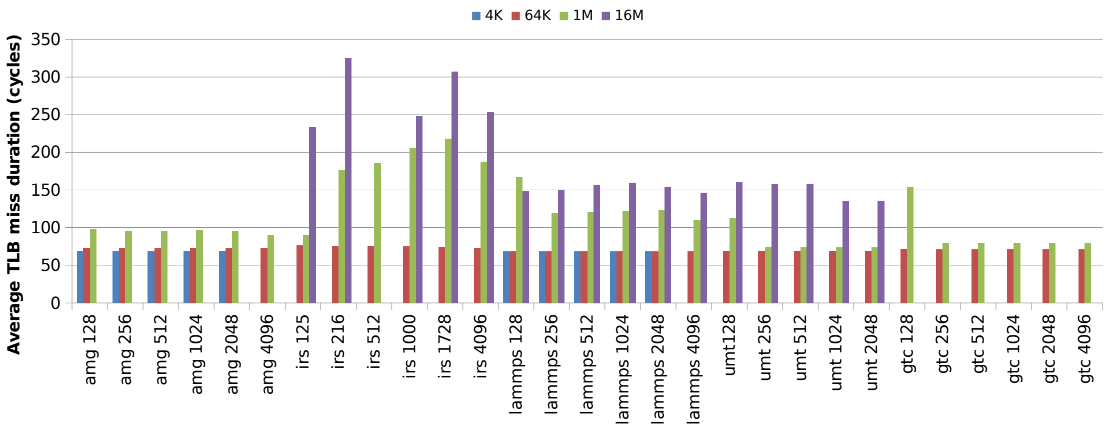


Figure 4.6: TLB miss handler execution time with varying page sizes and concurrent cores. The graph shows that the time required to perform a virtual-to-physical translation in CNK-TLB changes depending on the page size (the larger the page, the longer the execution time), especially with pages larger than 64KB. The TLB miss handler execution time influences the applications' overall overhead.Missing 4K page bars refer to experiments that could not complete due to the high overhead with small pages (see Table 4.3). Also, the 1M page experiment with IRS is missing due to the very low number of TLB misses to sample (less than 1 TLB miss on average).

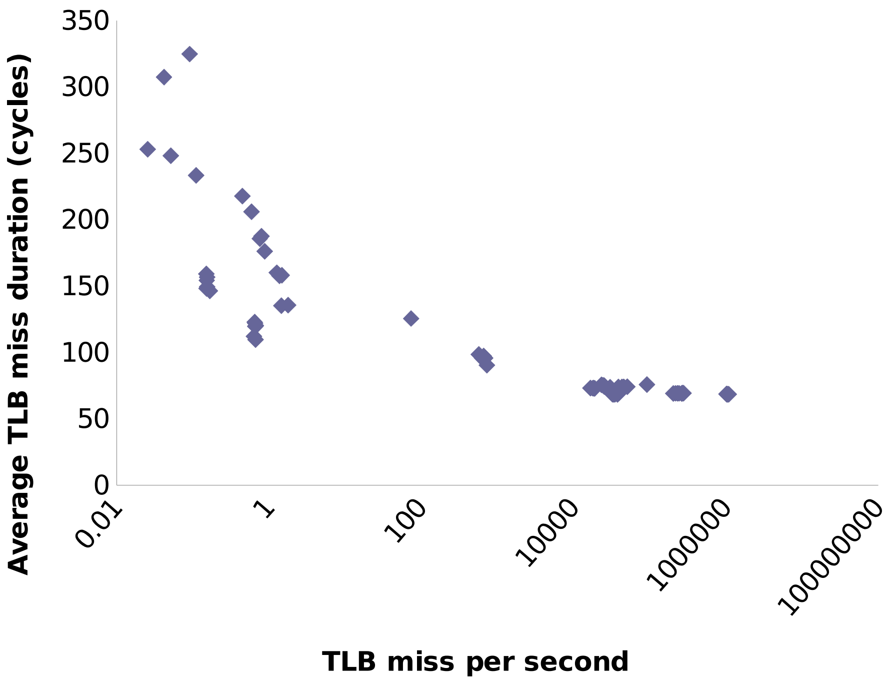


Figure 4.7: Empirical relationship between TLB misses/second and TLB handler execution time: when the TLB pressure is low (less than 1 TLB miss/second) the probability of cache misses increases with the result that the TLB handler takes longer to complete.

of TLB misses/second, the lower the average TLB miss handler execution time. When the number of TLB misses/second is high (i.e., more than 10K TLB misses/second), performing the TLB miss handler takes approximately 70 cycles on average. When the TLB miss frequency is low (less than 1 TLB miss/second) the overhead of determining the physical address requires more time (up to 325 cycles). Given the tlb miss handler is a software routine, it is subject to cache misses. With a low TLB miss frequency, the probability of data and instruction cache misses is higher, hence, on average, the TLB miss handler takes longer to resolve a virtual-to-physical translation.

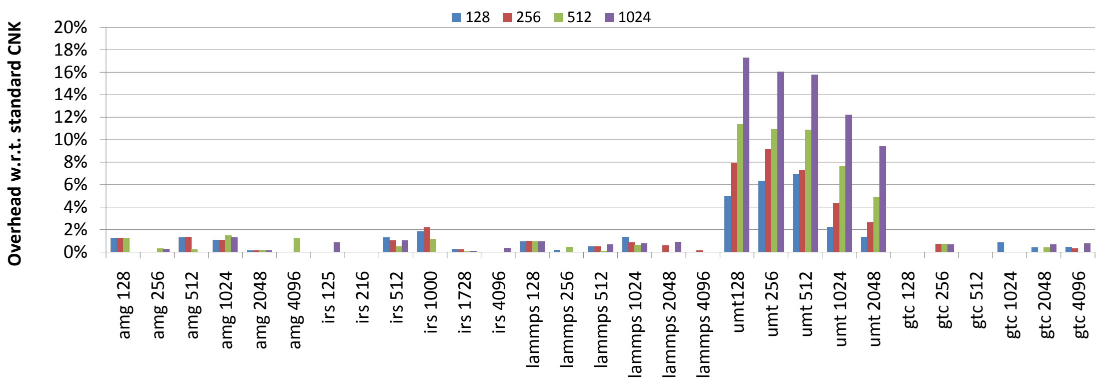
Figure 4.6 also explains why the performance slowdown experienced by IRS is higher than that of LAMMPS (Figure 4.5) even though LAMMPS poses an higher TLB pressure than IRS (Table 4.3): the average TLB miss handler execution time of IRS is higher than LAMMPS. An interesting observation is that, although the number of TLB misses/second is roughly constant at scale for most of the applications (e.g., AMG, IRS, LAMMPS, UMT, and GTC), the overhead introduced by TLB misses on a given application may vary. For example, the overhead experienced by AMG varies with the number of concurrent processes (Figure 4.5). For AMG with 64KB-pages both the number of TLB misses/second (16K-18.5K, see Table 4.3) and the TLB miss handler execution time (75 cycles, see Figure 4.6) are roughly constant at scale, however, the overall overhead increases from 1%, with 128 processes, to 5%, with 4096 concurrent processes. This is caused by *system noise resonance* [68], i.e., the effect of system noise for some applications is amplified at scale, depending on the criticality of TLB misses. In fact previous studies have shown that the effect of system noise at scale is much larger than the one measured on a single node [137, 75, 68, 148].

4.6.3 TLB noise injection

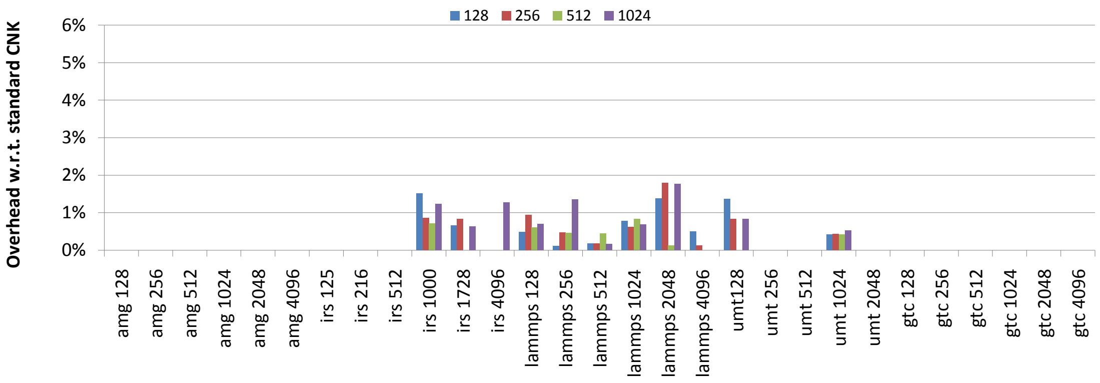
In the previous sections we analyzed the TLB pressure and the runtime overhead experienced by HPC applications when running on CNK-TLB, a modified version of CNK with replaceable TLBs that implements a range-checking TLB handler. In this Section we evaluate the effect of injecting TLB noise while running HPC applications. As described in Section 4.5.3, we introduce two different noise signature: a constant noise signature and a random signature uniformly distributed.

Constant signature The worst case scenario from the TLB noise point of view is one in which, at each TLB miss, each process of an application experiences the largest delay. We simulate this case by injecting a constant large delay of K cycles at every

TLB miss, with $K = 128, 256, 512, 1024$. Note that this delay sums to the average duration of the TLB handler of approximately 70 cycles.



(a) 1MB-pages



(b) 16MB-pages

Figure 4.8: Noise injection with constant noise signature. This graphs shows that the overall overhead of injecting TLB miss with a constant signature is below 2% with 16MB-pages for all the applications. With 1MB-pages, instead, UMT shows considerable overhead.

Figures 4.8a and 4.8b show the overall performance degradation experienced by the tested HPC applications when running on CNK-TLB with a constant delay injected at each TLB miss with respect to the standard CNK for 1MB and 16Mb pages, respectively. We do not report values for pages smaller than 1MB because, as Figure 4.5 shows, the overall overhead of TLB misses with 64KB-pages is already above our 2% threshold for some applications (e.g., UMT) even with the simple range-checking TLB miss handler. In these experiments, we vary the constant delay K from 128 cycles to 1024 cycles and the number of MPI processes. Figure 4.8a shows that most of the applications (AMG, IRS, LAMMPS and GTC) present an overall performance degradation below 2%, even with $K = 1024$ cycles. For these applications, and with 1MB pages, the TLB pressure is contained (see Table 4.3), thus, the applications can sustain large per-TLB miss overhead. UMT, on the other hand, poses a much higher TLB pressure: Even though the range-checking TLB handler evaluated in Section 4.6.2 introduces performance degradation below 2% for 1MB-pages, injecting a delay quickly degrades performance above 2%. With $K = 128$, UMT already experiences performance degradation up to 5%; with $K = 512$ the overhead quickly climbs up to 11.4%. The higher overhead for UMT seems to be a consequence of the number of TLB misses per second. Interestingly, the percentual overhead of UMT shows a decreasing trend as the number of cores increases.

Table 4.4 reports the execution times of the solving phase in CNK (no TLB misses), that is the baseline of this experiment. Although we run our experiments in weak scale mode, the total execution time for UMT increases from 19.51 seconds with 128 cores to 25.61 seconds with 2048. On the other hand, the direct overhead of TLB misses is constant (see Table 4.3 and Figure 4.6), thus we conclude that that overhead of TLB misses overlaps with other sources of performance degradation, causing the relative percentual overhead to decrease at scale.

Table 4.4: Execution time of the solving phase in seconds.

	Cores					
Apps	**128**	**256**	**512**	**1024**	**2048**	**4096**
amg	9.59	9.44	10.43	10.66	10.97	12.35
irs	27.37	33.46	42.36	58.48	72.13	114.45
lammps	66.05	64.96	65.48	66.44	65.94	67.02
umt	19.51	21.92	22.19	23.09	25.61	NA[1]
gtc	28.14	28.26	28.97	28.51	28.99	30.29

Although only one among the tested applications reports unsatisfactory results for 1MB pages, UMT is one of the applications that most closely represents future exascale applications, both in terms of complexity and implementation. The application is programmed with a mix of Fortran and C++ code, shared memory (OpenMP) and message passing (MPI) programming models and uses Python scripts to dynamically load computing modules. This experiment, thus, shows that 1MB pages may not be sustainable by future exascale systems, unless the per-TLB miss overhead is very contained, which may require hardware assistance. With 16MB pages, instead, the overall performance degradation is consistently below 2% for all applications, even with

[1]UMT uses half of the cores in each node. Hence using 1024 BG/P nodes we only have 2048 cores.

$K = 1024$ cycles. In fact, the performance measured are so close to the original CNK that the differences between two experiments are in the same order of the measurement error.

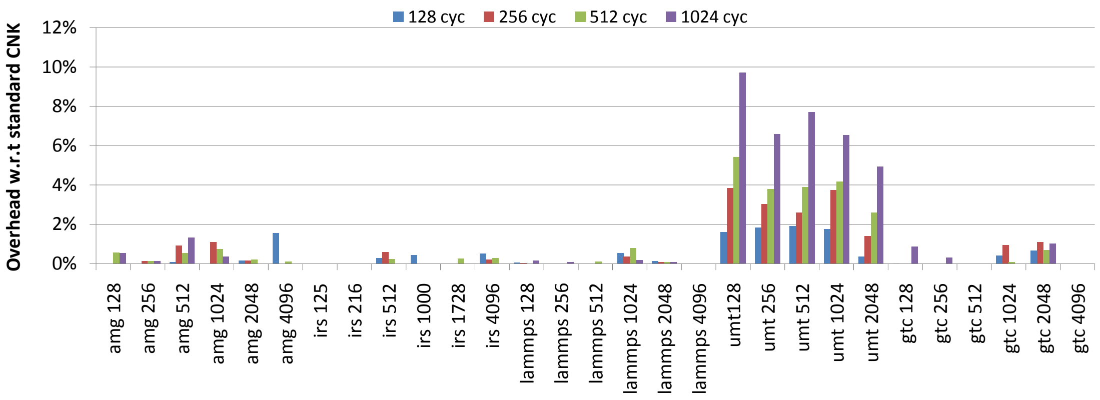
Random signature Although on a single node system noise is usually quantified 1-2%, performance degradation at scale can be much higher, especially if the system noise enters in *resonance* with the application [137, 75, 68]. If that happens, the application's performance is determined by the MPI process that experienced the largest delay, even though other processes experience smaller delays or no delay at all.

General purpose operating systems may introduce considerable system noise, some of which is related to the memory sub-system and to TLB miss handling. To simulate the random system noise generated by TLB miss handling we artificially introduce a random delay at every TLB miss. The random delay is uniformly distributed between 0 and T, with $T = 128$, 256, 512, and 1024 cycles. We use the core timebase register as source of random events and delay the TLB handler accordingly. The artificial delay sums to the duration of the range-checking TLB handler reproducing an handler with a minimum duration of approximately 70 cycles and a maximum duration of $70+T$.

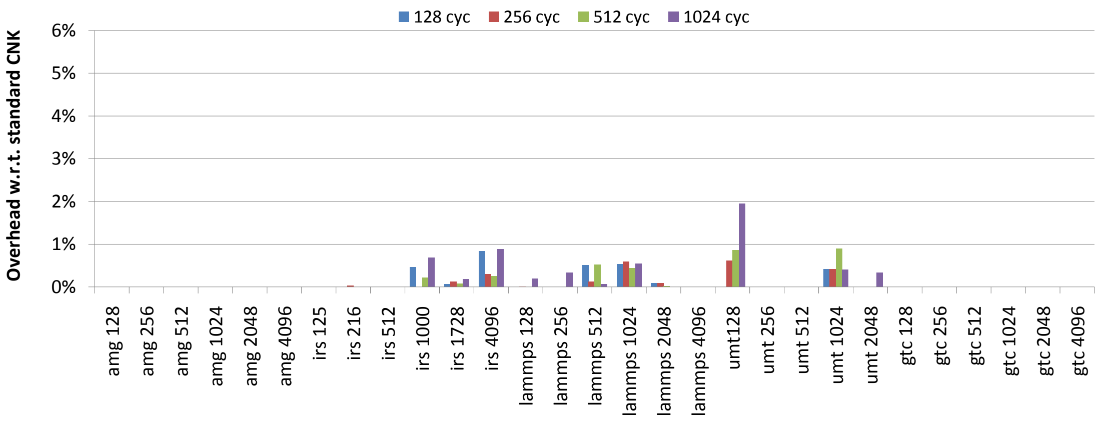
Figures 4.9a and 4.9b show the execution time overhead with respect to the standard CNK with no TLB misses. Note that, in Figure 4.9b, some benchmarks show 0% overhead because they do not present TLB misses in the solving phase with 16MB-pages (see Table 4.3).

Random delays up to 1024 cycles using 1M pages do not impact applications' performance significantly, except for UMT, as shown in Figure 4.9a. As for the case of constant noise signature injection, the overhead experienced by UMT with random noise signature is significant: with $T = 256$ cycles and 128 cores UMT shows 3.85% performance degradation, 9.70% with 2048 cores. The other applications all suffer a negligible overhead, as we consider overhead below 1% in the same order of the measurement error. Moreover, we observe that UMT performance degradation with respect to CNK decreases at scale, as reported for the constant signature noise injection.

These results confirm that 1M pages may affect applications performance, particularly for applications with a complex memory access patterns, such as UMT. On the other hand, 16MB-pages may introduce excessive memory fragmentation which, as future system will have less memory per core, may result in an unsustainable waste of resources. In these experiments we use pessimistic values of K and T (up to 1024 cycles), thus, it might not be necessary to move to 16MB pages. If that is necessary, however, reducing the number of TLB misses/second [29] or accelerating the TLB miss



(a) 1MB-pages



(b) 16MB-pages

Figure 4.9: Noise injection with random uniform noise signature. This graphs shows that the overall overhead of injecting TLB miss with a random uniform signature is below 2% with 16MB-pages for all the applications. With 1MB-pages, instead, UMT shows considerable overhead.

handler may reduce the overall overhead even with 1MB-pages.

4.7 Conclusions

Exposing the level of parallelism demanded by exascale supercomputers requires a richer system software ecosystem capable of supporting the novel programming models, technologies, and runtime systems that are emerging in the HPC community. To this extent, current petascale operating systems might need a fundamental re-design, especially for the memory sub-system part.

In this chapter we analyze the effects of TLB cache misses on current and future HPC applications. In order to isolate the effects of TLB misses from the other OS components and to provide an apples-to-apples comparison, we modified CNK, a noiseless OS for Blue Gene systems that implement a fixed TLBs design, by adding support for replaceable TLBs. We implemented several TLB miss handlers: First we designed a low-overhead, range-checking algorithm that resolves virtual-to-physical mappings with minimum overhead. This algorithm causes the minimum overhead to translate a virtual-address into a physical-address. Second, to evaluate the effects of a more complex memory sub-system (e.g., paging), we varied the duration of the TLB miss handler by injecting artificial delays with random and constant signatures. Then, we studied the effects of varying the page size from 4KB to 16MB and the scalability implications of taking TLB misses.

We consider 2% as the maximum performance degradation that a programmer would be willing to pay in order to have richer system software ecosystem. Our experiments, performed with representative applications from the LLNL Sequoia and ANL benchmarks on a BG/P system with up to 1024 nodes (4096 cores), show that, for most of the applications, the overhead of TLB misses is below 2% even with 1MB-pages and with extremely pessimistic delays (1024 cycles). For more complex applications, such as *UMT*, that poses very high pressure on the TLB, 1MB-pages may not satisfy our performance requirements, especially if the TLB miss handler execution time is large. Such complex applications show overall overhead below 2% with 16MB-pages even with large artificial delays.

However, 16MB-pages may introduce higher memory fragmentation, and in a scenario with less memory per core, this could be a major issue. In this case, hardware and software techniques should be used to reduce the TLB miss overhead and minimize performance degradation even with smaller page sizes. In the specific case of BlueGene systems, the most recent version (BG/Q) provides an MMU to speed up virtual-to-physical mapping.

Part III

Runtime System Scalability

Chapter 5

Scalable Runtimes for Distributed Memory Systems

The previous part of the thesis focused on the scalability of operating systems. In this part we focus on the scalability of the upper layer of system software: the runtime system. Traditional high performance applications used a fairly simple runtime system based on a message passing programming models. As mentioned in the introduction, the advent of more complex hardware fostered the design of new programming model and intelligent/adaptive runtime systems. Applications are changing too, moving from regular to irregular computation and data access patterns (e.g. graph-based applications). Performance of large scale high performance systems is becoming more and more a function of data movement rather than floating point computation. A new class of applications (so-called irregular applications) are gaining increasing interest, because of the important nature of the problems that they address (data intensive applications). In this chapter, we describe the design and implementation of a specialized runtime system and programming model to efficiently execute irregular applications on commodity clusters.

5.1 Summary

Bioinformatics, big data science applications, complex network analysis, community detection, data analytics, language understanding, pattern recognition, semantic databases and, in general, knowledge discovery constitute a new generation of irregular high performance computing applications [48]. The size of the datasets in these applications has already surpassed a petabyte, and is exponentially growing. Only by exploiting large clusters which enable in-memory processing of the entire datasets, there may be a chance to scale both the size and performance of these applications.

These applications exploit pointer-based data structures such as graphs, unbalanced trees or unstructured grids, which exhibit poor spatial and temporal locality, and have fine-grained, unpredictable data accesses, which lead to ineffective utilization of the available memory and network bandwidth [169]. These applications are also inherently parallel, because they can potentially spawn a concurrent activity for each element of the datasets (e.g., each vertex or each edge in a graph). However, their datasets are difficult to partition without generating load imbalance across the nodes of a cluster, because the elements often are highly interconnected (e.g., power law graphs). Furthermore, they often present high synchronization intensity, because multiple threads often need to access and/or update the same elements.

Modern clusters employ processors with advanced cache architectures and rely on data locality, regular computations and bulk communication to achieve high performance. Implementing irregular applications on these machines requires a significant and highly application-specific optimization effort. However, as high as the effort might be, it often results in poor system resource utilization. The distributed memory architecture of modern clusters further complicates their design, forcing the developers to search for efficient ways to partition datasets and minimize communication overheads. Different approaches have been proposed for efficiently executing large scale irregular applications. They include custom supercomputers, such as the Cray XMT [66], which implements a multithreaded processor to tolerate memory and network access latencies, a scrambled global address space across nodes and full/empty bits associated with each memory word for fine-grain synchronization. However, the reduced market for custom architectures makes them expensive to produce and maintain. On the other hand, distributed graph libraries based on extensions to the MapReduce paradigm are now widely used [80, 111]. Nevertheless, they are not general enough for supporting other irregular applications beside graphs, and do not scale when graphs have complex structures.

The Partitioned Global Address Space (PGAS) programming model seems to be a promising alternative to develop applications with a shared memory abstraction using distributed memory clusters, without neglecting the concept of locality. However, current PGAS languages and libraries target regular applications, where communication across nodes happens using large block transfers. Furthermore, they normally use the Single Program Multiple Data (SPMD) control paradigm, where the program starts at the beginning with multiple, identical parallel processes. The SPMD model does not cope well with the dynamic and fine-grained nature of the parallelism in irregular applications. Parallelism in irregular applications is highly dynamic and unbalanced, in many cases requiring dynamic spawning of new concurrent activities as new data is “discovered”.

In this work we introduce **GMT** (Global Memory and Threading library), a custom runtime library that enables efficient execution of irregular applications on commodity clusters. GMT integrates the PGAS locality-aware global data model with loop parallelism, commonly used in single node multithreaded environments. GMT enables the development of applications with large datasets that span multiple nodes without requiring domain decomposition or data partitioning. GMT exposes to application developers a very simple application programming interface (API) that provides operations such as: allocating and deallocating memory in the global, logically shared address space; reading and writing data into the global memory; performing synchronization operations on any global memory location. The parallelism is identified through parallel loop constructs, and multiple nested loops are efficiently executed. These constructs enable the expression of the large amount of fine grained parallelism typically found in irregular applications. Lightweight threading is the main mechanism used to tolerate remote data access latencies, by context switching tasks that are waiting for remote operations. GMT supports millions of lightweight, user-level tasks. Thousands of these tasks are multiplexed on each available core of the cluster.

In GMT each CPU core executes a specialized thread. There are three classes of specialized threads: *workers*, which execute the tasks composing the application; *helpers*, which manage PGAS operations and synchronization; and communication server threads, which perform the actual remote data transfers. At the core of GMT resides the key concept of remote request aggregation (also known as network message coalescing): composing multiple application-level fine-grained requests to improve network bandwidth utilization. GMT features multiple levels of aggregation: requests are first aggregated in the local core and subsequently aggregated at node level. Finally, GMT runs on any homogeneous cluster based on x86 processors, which supports MPI.

We compare GMT to other PGAS models (UPC on GASNet), to hand-optimized MPI code, and to custom machines (Cray XMT) on a set of typical kernels of large scale irregular applications: Breadth First Search (BFS), Random Graph Walk (RGW) and Concurrent Hash-Map Access (CHMA). We demonstrate performance orders of magnitude higher than other solutions for commodity clusters, and competitive performance compared to custom systems. We show that GMT enables high scalability in performance and dataset size, as additional nodes are added to the system.

The chapter is organized as follows. Section 5.3 discusses relevant related work. Section 5.4 describes the programming model and GMT's API. Section 5.5 illustrates the architecture of the runtime system, detailing the design choices behind the most relevant software components. Section 5.6 discusses our experimental evaluation. It initially characterizes the communication performance of the library, and then compares the performance obtained on the three irregular kernels with the other implementations.

5.2 Experimental environment

Tables 6.1,6.2 and 5.3 show the experimental environment used in this chapter. The tables include the hardware systems, the communication libraries, and the benchmarks used for the experiments.

CPU	AMD Opteron 6272 processors (codename "Interlagos") at 2.1 Ghz (2 sockets - 8 cores per socket)
Memory	64 Gigabyte of DDR3 RAM
Network	Infiniband QDR (40 Gb/s)
System size	up to 128 nodes (4096 cores)
Operating System	Linux 2.6.32
Communication Library	GMT and OpenMPI 1.5.4
Benchmarks	GMT microbenchmarks, Breadth First Search, Graph Random Walk and Concurrent Hash Map Access

Table 5.1: Experimental environment - PNNL Institutional Computing (PIC) cluster system

CPU	500 MHz Single 64-bit Cray Threadstorm Processor (128 threads per processor)
Memory	1 TB of total RAM (shared memory)
Network	Seastar-2
System size	up to 128 processors
Operating System	Linux and MTK
Communication Library	custom pragmas
Benchmarks	Breadth First Search

Table 5.2: Experimental environment - PNNL Cray XMT system

CPU	4 AMD Opteron processors "Magny-Cours" at 2.3 GHz (4 sockets - 12 cores per socket)
Memory	256 GB of DDR3 RAM
Network	-
System size	single node
Operating System	Linux
Communication Library	OpenMP
Benchmarks	Breadth First Search

Table 5.3: Experimental environment - PNNL AMD Many-core workstation

5.3 Related Work

Efficiently executing large-scale irregular applications has been a significant research topic for a while. Due to the complexity in managing (and partitioning) the datasets, irregular applications are better suited to shared memory systems. However, even if modern shared memory multiprocessors reach memory sizes in the order of terabytes, the datasets of relevant real world scientific and knowledge discovery problems may easily surpass these sizes. Executing irregular applications on clustered systems may allow scaling both the dataset size and the performance. Nevertheless, current high performance clusters are optimized for regular applications, and based around distributed memory hardware. Several approaches have been proposed to execute irregular applications on distributed memory systems, ranging from custom hardware systems to full software infrastructures.

The Tera MTA [150] and its successors, Cray MTA-2 [15], XMT [66] and uRiKa [51] are the most relevant examples of custom multi-node supercomputers for irregular applications. They use simple, highly multithreaded (up to 128 hardware threads), cacheless VLIW processors that support a global address space in hardware, interconnected with the high-performance Cray Seastar-2 network. The large number of threads allows tolerating both local and remote memory access latencies. The address space is scrambled across nodes at a fine granularity (64 bytes on the XMT), reducing the hot-spot occurrence and obtaining more uniform data access times. Every memory word is associated with full/empty bits that provide fine grain synchronization. Applications for this category of machines are written in a shared memory paradigm. A custom C/C++ compiler semi-automatically parallelizes the code exploiting `pragmas` and analyzing parallel loop nests, thus enabling the data-parallel mapping of the loops' iterations to threads.

Another approach to execute irregular applications with large datasets uses modern shared memory nodes with Solid State Disks (SSDs). SSD enables the management of datasets larger than the available Random Access Memory, significantly reducing penalties for swapping. However, this approach only allows to scale the dataset size, but not the performance. Our approach, instead, aims at scaling both performance and size as more nodes are added to a clustered system. Furthermore, by supporting latency tolerance and data aggregation, our runtime could also enable better exploitation of the mass storage.

There are several types of software approaches that have been proposed to enhance the performance and ease the development of irregular applications on distributed memory machines. Multipol [45] is a library of distributed data structures for irregular problems. It includes stateful structures (hash tables, sets, trees), which exploit replication, partitioning and software-controlled caching for obtaining locality, and scheduling structures (various kinds of queues) for load balancing. The CHAOS/PARTI [52] runtime support library is a set of software primitives that couple partitioners to the application programs, remap data and partition work among processors, and optimize interprocessor communications. These libraries are mainly targeted at optimizing the data partitioning and at reducing communication overheads across processors/nodes through caching mechanisms. The work-to-processors mapping is mostly hand-managed. Like these libraries, our runtime system provides a global address space and optimizes the communication. However, rather than exploiting caching, GMT improves effective bandwidth using data aggregation, and also integrates latency tolerance mechanisms via software multithreading. Finally, exploiting more modern multicore architectures, our library delegates different operations to different threads.

The Partitioned Global Address Space programming model (PGAS) provides a more general framework for irregular applications, by realizing an abstracted shared address space across distributed memory systems, without neglecting data or thread locality. The PGAS programming model is implemented in languages and libraries such as: Unified Parallel C (UPC) [60, 49, 167], Co-Array Fortran (CAF) [132, 93], the Global Arrays (GA) Toolkit [130] and others. These programming models rely on communication runtime libraries that manage the data exchanges between distributed address spaces, amongst them GASNet [32] and ARMCI [128]. GASNet serves as a communication runtime to several PGAS languages including UPC, Titanium [168], and Co-Array Fortran [93]. X10 [47] is an object oriented parallel language based on the Asynchronous PGAS (APGAS) model, which extends the PGAS model with the concepts of places and asynchrony via tasks. Chapel [46] is a parallel programming language for distributed memory supercomputers based on ideas from HPF, ZPL and the extensions introduced in the MTA/XMT compiler. Similarly to X10, it supports asynchronous task execution and reasoning about locality via the *locale* abstraction.

Our runtime is also based around the Partitioned Global Address Space concept, however current PGAS models mainly target regular applications, and except for X10 and Chapel, employ SPMD control models. Instead, we specifically target irregular applications, and a simple task-based, parallel loop control model. Our runtime implements techniques such as latency tolerance through lightweight multithreading, data aggregation and (currently intra-node) load balancing, to cope with the behavior of irregular applications, and hides locality to the application developers. We do not aim at providing a full parallel language, but rather a set of simple primitives to implement irregular applications on distributed memory systems by adopting a shared memory abstraction. Code translators and parallel extensions to other languages such as C could use GMT as their target runtime layer.

Grappa [126] is a runtime which integrates a PGAS programming model and multithreading for latency tolerance, targeted at increasing performance of graph crawling on commodity x86 clusters. Compared to our approach, it employs a substantially different architecture: it is based on GASNet, it has limited support for data aggregation, and it does not exploit thread specialization.

Active Pebbles [166] includes both a programming model for fine-grained data-driven computations and an execution model which maps the fine-grained expression to an efficient implementation. The programming model exploits the concepts of pebbles and targets. Pebbles are light-weight active messages that operate on targets. Targets are the binding of a data object with a message handler, through a distribution object, and form a global address space. The execution model includes message aggregation, active routing, message reduction and termination detection (because each pebble can trigger new communications). Our objective is not to define a new programming model, but rather to provide a simple API to implement irregular applications, supporting loop parallelism. Our runtime, besides aggregation, also exploits thread specialization.

Charm++ [96] is a parallel programming system based on a programming model that exploits message-driven objects (*chares*). When a programmer invokes a method on an object, the runtime sends a message to the object, which may be local or remote, starting asynchronous operations. The runtime can adaptively assign chares to processors during program execution, and supports latency tolerance by switching from blocked processes to others. With respect to Charm++, our runtime explicitly targets irregular applications, and for this reason supports much finer task granularity for latency tolerance and data aggregation.

ParalleX (PX) [70] is an experimental execution model that exploits a split-phase multithreaded transaction distributed computing methodology, decoupling computation and communication. ParalleX supports fine-grain multithreading, global address space, overlapping of communication and computation and is able to move work to the data. High Performance PX (HPX) is a runtime system based on the ParalleX execution model. Although ParalleX also targets irregular workloads, it is significantly more complex in scope than GMT. As such, it is also more complex to use and the current HPX runtime is still incomplete, missing elements such as data aggregation, and developed with an approach focused to provide features, rather than performance, first.

OCR [44] is a framework to explore new methods of high-core-count programming. The initial focus is on HPC applications and its goal is to improve power efficiency, programmability, and reliability while maintaining performance. OCR provides event-driven tasks, events, memory data blocks , machine description facilities, and more.

There are several libraries for graph processing on distributed systems. Among them, the most widely used are Pregel [115], Giraph [3] and GraphLab [111]. They all exploit bulk synchronous parallel models (usually map-reduce) through vertex programs that run on each vertex and interact along edges. Interactions either use messages (Pregel and Giraph) or shared states (GraphLab). However, they only aim at solving graph traversal through a specific API. Our library, instead, targets a wider class of irregular applications.

Although map-reduce frameworks have been applied to several irregular problems [107], they cannot deal with the more general cases where computation of values depends on previously computed values [114]. Examples are, for example, queries executed on semantic graph databases through graph pattern matching operations, which may dynamically generate new (parallel) searches or stop the execution depending on the outcome of part of the query.

5.4 Programming model and API

GMT aims at providing an effective way to program large-scale irregular applications on commodity clusters. Considering the three dimensions of productivity, performance and generality, we designed GMT to favor productivity and performance over generality. We designed a productive programming model for irregular applications, and rely on the runtime system to enable high performance and scalability.

In our view, the programmer should not think about the details of the underlying distributed memory machine, but rather in terms of an abstract distributed memory system. To this aim, we borrow well-know programming model concepts commonly used in shared memory systems.

Table 5.4 summarizes the primitives currently provided by GMT. In the following sections we describe the characteristics of GMT programming model and then we show an example of an irregular kernel implemented with GMT's API.

Primitive	**Functionality**
gmt_array gmt_alloc(size, locality)	Allocates a gmt_array with the specified data distribution (partitioned, remote, local)
gmt_free(gmt_array)	Deallocates a previously allocated gmt_array
gmt_putNB(gmt_array, offset, *data, size)	Puts a local buffer into the indicated gmt_array starting at the specified offset (non blocking)
gmt_putValueNB(gmt_array, offset, value, size)	Puts a value into the gmt_array at the specified offset (non blocking)
gmt_getNB(gmt_array, offset, *data, size)	Gets a portion of a gmt_array starting from offset into a local buffer (non blocking)
gmt_waitCommands()	Waits for completion of previously issued non-blocking operations
gmt_put(gmt_array, offset, *data, size)	Blocking put
gmt_putValue(gmt_array, offset, value, size)	Blocking putValue operation
gmt_get(gmt_array, offset, *data, size)	Blocking get
gmt_atomicAdd(gmt_array, offset, value, size)	Atomically adds a value to the value contained in a gmt_array at the specified offset
gmt_atomicCAS(gmt_array, offset, oldValue, newValue, size)	Exchanges a value with the value contained in a gmt_array at the specified offset. Returns the old value
gmt_parFor(tot_iters, chunk_size, *tasks, *args, locality)	Spawn tasks that execute the iterations, up to the total number of iterations, and takes as input the specified argument buffer. Tasks are spawned on all the allocated nodes of the system, locally or remotely, and execute chunk_size iterations per task.

Table 5.4: GMT API summary

5.4.1 PGAS communication model

The PGAS communication model simplifies the application developers' task of partitioning data structures across nodes. Data is partitioned between global data and local data. The programmer allocates the data structures, mostly arrays, in a virtual global address space, and accesses them through *get* and *put* operations (see Table 5.4). Global data structures are identified by handlers that are passed to the various GMT primitives. Global data is moved into the local space to be manipulated and then written back into the global space. This approach enables the programmer to ignore the actual memory address and cluster node where the data is allocated.

5.4.2 Loop parallelism program structure model

In GMT, the programmer express parallelism through a parallel loop construct (*gmt_parFor()* in Table 5.4), a parallel control model typical of shared memory paradigms. In contrast with the SPMD model, this model allows efficiently creating tasks. In GMT a *task* is a user-defined function executed for several iterations using the *gmt_parFor()* construct.

The parallel loop construct enables creation of new tasks from iterations of loops over independent individual structure elements (e.g., parallel loops over all vertices or edges of a graph).The application developer can specify how many iterations of the original loop to assign to each task (*chunk_size*), but the runtime is also capable of dynamically detecting if the same processing entity should execute more iterations for load balancing purposes. In the current implementation, the calling task is suspended until all the iterations of the parallel loop are completed. Finally, GMT also supports nested parallel loops, enabling programming patterns such as recursive parallel constructs.

5.4.3 Explicit data and code locality management

The GMT allocation primitive offers the ability to control the data distribution through distribution strategies. The *GMT_ALLOC_PARTITION* strategy allocates data in a block distributed manner, so that it is uniformly distributed across all the nodes. The *GMT_ALLOC_LOCAL* strategy allocates data only on the memory of the local node. Finally, the *GMT_ALLOC_REMOTE* strategy allocates data on all other nodes except the one that executes the primitive. GMT does not expose the physical location of data to the programmer, to avoid explicit management of data pointers and node ranks.

Analogously, GMT task creation policies (*GMT_SPAWN_PARTITION*, *GMT_SPAWN_LOCAL* and *GMT_SPAWN_REMOTE*) control the locality of the tasks created by a parallel loop. The programmer only controls the locality policy, while the runtime takes care of transparently mapping the tasks to the available cluster resources (i.e., processor cores).

5.4.4 Blocking and Non-blocking semantics

GMT communication primitives feature both blocking and non-blocking semantics. When using the blocking flavor of the primitives, the task suspends until the operation effectively completes. When the function of a blocking operation returns, the termination of the operation is guaranteed, for both remote and local operations. When using the non-blocking flavor of the primitives, the task continues, and the order of operations is not guaranteed. When the code calls *gmt_waitCommands()*, the task is suspended, until the runtime completes all the pending non-blocking operations. For performance reasons, given the fine-grain nature of communication operations, *gmt_waitCommands()* does not allow waiting for a specific non-blocking operation.

5.4.5 Explicit synchronization

The programmer explicitly specifies synchronized access to global data structures. GMT provides atomic operations such as *gmt_atomicCAS()* or *gmt_atomicAdd()* (see Table 5.4) to enable the implementation of global synchronization constructs.

5.4.6 Example

Figure 5.1 shows the code of a simple sequential Breadth First Search (BFS). The code shows the allocation of the graph data structure in Compressed Sparse Row (CSR) form (represented as 1D arrays of offsets and edges), of the current and next iteration queues, and of the map for visited vertices. The queues use the first location to store the total number of elements in the queue. The initialization procedure initializes the graph data structures and puts the root vertex in the exploration queue. The main loop analyses the vertices in the exploration queue, and goes through all the neighbors of each one. If the algorithm did not previously visit a neighbor, it is marked as visited and added to the queue for the next iteration. After visiting all the vertices in the current iteration queue, the algorithm exchange pointers and swaps the queue of vertices to explore for the next iteration as the current one. The code also shows the memory deallocation operations.

Figure 5.2 shows the parallel code of this implementation under GMT. The data structures are now allocated in the GMT PGAS memory space, using the *gmt_alloc* construct. All the data structures are allocated with the *GMT_ALLOC_PARTITION*

```
int main(){
    // n = number of total vertices
    // e = number of total edges
    uint64_t *offsets = malloc((n+1)*sizeof(uint64_t));
    uint64_t *edges   = malloc(e*sizeof(uint64_t));
    uint64_t *q       = malloc((1+n)*sizeof(uint64_t));
    uint64_t *qnext   = malloc((1+n)*sizeof(uint64_t));
    uint8_t  *map     = malloc(n*sizeof(uint8_t ));

    init(offset, edges, q, qnext, map);
    uint64_t q_size=q[0];
    while (q_size!=0) {
        int vid;
        for (iterId = 1; iterId<q_size; iterId++) {
            uint64_t v=q[iterId];
            int i;
            for (i=offsets[v]; i<offset[v+1]; i++) {
                uint64_t neighbor=edges[i];
                if (map[neighbor]==0) {
                    map[neighbor]=1;
                    qnext[++qnext[0]]=neighbor;
                }
            }
        }
        uint64_t *qtmp = q;
        q=qnext;
        qnext=qtmp;
        q_size=q[0];
        qnext[0]=0;
    }
    free(offsets); free(edges); free(q); free(q_next); free(map);
}
```

Figure 5.1: Sequential queue-based BFS implementation

```
typedef struct args_bfs_tag {
    gmt_data_t g_offset;
    gmt_data_t g_edges;
    gmt_data_t g_map;
    gmt_data_t g_q;
    gmt_data_t g_qnext;
} args_bfs_t;

int main(){
    // n = number of total vertices
    // e = number of total edges
    gmt_data_t g_offsets = gmt_alloc((n+1)*sizeof(uint64_t),GMT_PARTITION);
    gmt_data_t g_edges   = gmt_alloc(e*sizeof(uint64_t),GMT_PARTITION);
    gmt_data_t g_q       = gmt_alloc((1+n)*sizeof(uint64_t),GMT_PARTITION);
    gmt_data_t g_qnext   = gmt_alloc((1+n)*sizeof(uint64_t),GMT_PARTITION);
    gmt_data_t g_map     = gmt_alloc(n*sizeof(uint8_t),GMT_PARTITION);

    init(g_offset, g_edges, g_q, g_qnext, g_map);
    uint64_t g_q_size;
    gmt_get (g_q,0*sizeof(uint64_t),&g_q_size,sizeof(uint64_t));

    while(g_q_size!=0) {
        args_bfs_t args;
        args.g_edges  = g_edges;
        args.g_map    = g_map;
        args.g_offset = g_offset;
        args.g_qnext  = g_qnext;
        args.g_q      = g_q;

        gmt_parFor(g_q_size,1,body_bfs,&args,sizeof(args_bfs_t),GMT_PARTITION);

        gmt_data_t g_qtmp = g_q;
        g_q = g_qnext;
        g_qnext = g_qtmp;

        gmt_get(g_q,0*sizeof(uint64_t),&g_q_size,sizeof(uint64_t));
        gmt_putValue(g_qnext,0*sizeof(uint64_t),0,sizeof(uint64_t));
    }

    gmt_free(g_offsets); gmt_free(g_edges); gmt_free(g_q);
    gmt_free(g_qnext); gmt_free(g_map):
}

void body_bfs (uint64_t iterId, void *args) {
    args_bfs_t *largs   = (args_bfs_t*) args;
    gmt_data_t g_offset = largs->g_offeset;
    gmt_data_t g_edges  = largs->g_edges;
    gmt_data_t g_map    = largs->g_map;
    gmt_data_t g_q      = largs->g_q;
    gmt_data_t g_qnext  = largs->g_qnext;

    uint64_t v = 0, offsets[2];

    gmt_get(g_q,(iterId+1)*sizeof(uint64_t),&v,sizeof(uint64_t));
    gmt_get(g_offset,v*sizeof(uint64_t),&offsets,2*sizeof(uint64_t));

    uint32_t i;
    uint64_t neighbor;
    for (i=0;i<offsets[1]-offsets[0];i++) {
         gmt_get(g_edges,(offsets[0]+i)*sizeof(uint64_t),&neighbor,sizeof(uint64_t));
         if(gmt_atomicCAS(g_map,neighbor*sizeof(uint8_t),0,1, sizeof(uint8_t))== 0) {
            uint64_t Qnext_N=gmt_atomicAdd(g_qnext,0*sizeof(uint64_t),1,sizeof(uint64_t));
            gmt_putValue(g_qnext,(Qnext_N+1)*sizeof(uint64_t),neighbor,sizeof(uint64_t));
        }
    }
}
```

Figure 5.2: Parallel BFS implementation with GMT

strategy, which distributes them in contiguous blocks of the same size on all the allocated nodes. The graph data structure and the queues are stored in the global memory space. For this reason, the algorithm needs to retrieve the number of elements in the queue using a get operation. Control flow is sequential from the main task until the parallel loop construct is encountered, immediately after its completion control goes back to the main task again. The main loop, which goes through all the elements in the current iteration queue, is substituted with a parallel for (*gmt_parFor*) construct. The parallel for construct launches as many tasks as the number of vertices in the queue, corresponding to the iterations of the for loop. The *gmt_parFor* construct uses the PARTITION iteration distribution strategy, which allows spawning tasks on all the executing nodes. A `struct`, containing the references to the global data structures is used to communicate their location to the tasks. The task itself is a function, which takes as input the structure of parameters, as well as an iteration identifier (iterId). The iterId allows addressing the global data structures in a data-parallel manner through offsets and works as a task identifier. Data in the global data structures is accessed through *get* and *put* operations, by using offsets derived from the iterIds. The parallel implementation uses atomic compare and swap to mark vertices not previously explored and atomic addition to reserve slots and add them to the queue for the next iteration. References to the queues allocated in global memory are exchanged like pointers. The example demonstrates the simplicity in implementing applications with GMT. Porting from sequential or shared memory implementations is straightforward, and the developer does not need to worry about data partitioning or domain decomposition.

5.5 Runtime architecture

We built GMT around three main "pillars": *global address space*, latency tolerance through fine-grained software *multithreading*, and remote data access *aggregation* (also known as coalescing). As previously discussed, global address space support relieves application developers from having to partition data sets as well as having to orchestrate communication. Message aggregation (coalescing) maximizes network bandwidth utilization, despite the small data accesses typical of irregular applications. Fine-grained multithreading enables applications to perform useful work while communication is in progress, hence hiding latencies for remote data transfers as well as the added latency for aggregation. We followed a bottom-up approach in the design and implementation of GMT. We identified the basic building blocks and, for each one, we evaluated the performance of the alternative design points, selecting the best solutions. The follow-

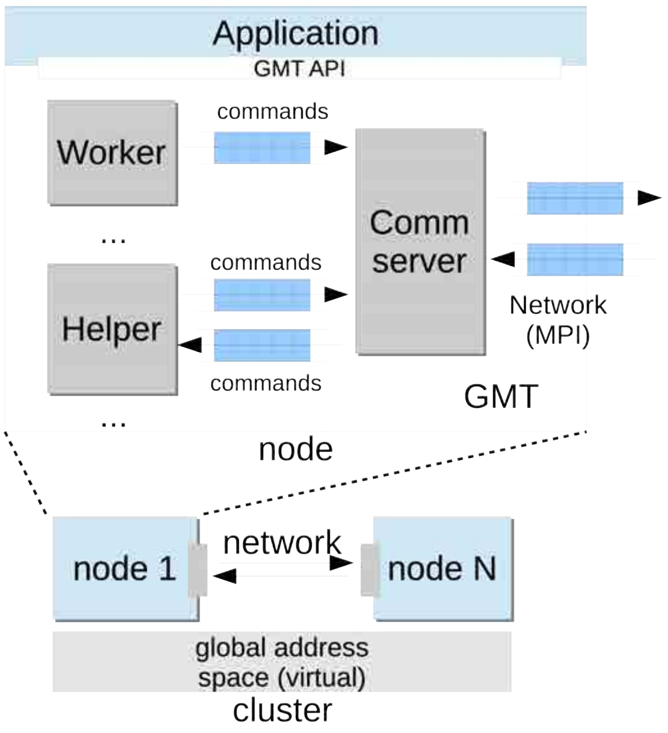


Figure 5.3: Architecture overview of GMT

ing sections describe the most significant components of the runtime and explain the rationale behind them.

For exploring the design choices of GMT's building blocks, and for the overall experimental evaluation, we employed Pacific Northwest National Laboratory's Olympus supercomputer, listed in the TOP500 [1]. Olympus is a cluster of 604 nodes interconnected through a QDR Infiniband switch with 648 ports (theoretical peak of 4GB/s). Each Olympus' node features two AMD Opteron 6272 processors (codename "Interlagos") at 2.1 Ghz and 64 GB of DDR3 memory clocked at 1600 Mhz. Each socket hosts 8 processor modules (two integer cores, one floating point core per module) on two different dies, for a total of 32 integer cores per node. A module includes a L1 instruction cache of 64 KB, two L1 data caches of 64 KB, and a 2 MB L2 cache. Each 4-module die hosts a shared L3 cache of 8 MB. Dies and processors communicates through HyperTransport.

5.5.1 Overview

GMT targets commodity clusters, composed of nodes with multicore processors (such as the latest AMD Opteron or Intel Xeon), interconnected with modern, fast networks such as Infiniband or the Cray high performance interconnects (Gemini, Aries). Figure 5.3 illustrates the high level design of GMT. GMT realizes a virtual global address space across the nodes of the cluster. Each node executes an instance of GMT and the various instances communicate through *commands*. Different types of commands exist for GMT operations such as global data read/write, synchronization and thread management. Commands may also include data movement (e.g., *gmt_put()* and *gmt_get()*). An instance of GMT, executing in one cluster node, includes three different types of *specialized* threads:

- *Worker*: executes the application code, partitioned in task, and generates requests, in form of commands, directed towards both the local node and the remote nodes.
- *Helper*: manages global address space and synchronization, handles incoming requests and generates the related outgoing replies, in form of commands.
- *Communication server*: communication endpoint on the network, manages incoming and outgoing communication at the node level. Workers and Helpers send commands to the Communication Server, which forwards them to the remote nodes.

A GMT node includes multiple workers and helpers, but only a single communication server. We implemented the specialized threads as *POSIX* threads. Each thread is pinned on a core.

5.5.2 Communication

A fundamental design point for GMT is the choice for the underlying communication library. GMT does not use communication libraries that already provides PGAS primitives, such as GASNet [32], because their implicit communication management mechanisms do not provide message aggregation. Hence, we decided to implement our own PGAS primitives, designing them with message aggregation from the ground-up. For this reason, the only requirement is a message passing interface, optimized for high

message size	32 proc. no threads	1 proc. 1 thread	1 proc. 2 threads	1 proc. 4 threads
16B	9.63	4.22	2.73	0.77
32B	19.54	9.63	5.70	1.58
64B	39.05	19.54	10.99	3.12
128B	72.26	39.05	19.73	6.22
16KB	2806.94	1924.98	646.52	269.63
32KB	2806.95	2250.15	892.80	469.40
64KB	2815.01	2559.50	794.60	566.09
128KB	2835.98	2709.07	1042.01	564.87

Table 5.5: Transfer rates in MB/s between two nodes with varying thread and process number.

bandwidth. We selected MPI, because it is the de-facto standard for message passing interfaces, and supports the broadest variety of architectures.

We then determined the number of Communication Servers required to maximize node-to-node bandwidth with MPI. We analyzed several combinations of MPI processes and threads per node to determine the highest bandwidth. Table 5.5 presents a comparison of the transfer rates between two Olympus' nodes, when transferring a large number of messages from one node to the other and waiting the acknowledge from the receiver for every 4 messages.

We used a slightly modified version of OSU Micro-Benchmarks 3.9 [6] (in order to support multithreading) to compare MPI (OpenMPI 1.5.4) with multiple processes (32 on the same node) and MPI (MVAPICH 1.9b) with one, two and four threads.[1] MPI with multiple threads per process exhibits low transfer-rates.

The best performance is obtained using large messages with 32 processes per node. Nonetheless, because of the complexity and the high memory foot-print of managing different address spaces we consider the latter an unfeasible solution for GMT. As shown in Table 5.5 transfer rates are particularly sensitive to the message size. Even if we are showing results for Olympus, we observed similar MPI behaviour with other processor architectures and network interconnects. These results drove our decision to design GMT with a single communication server, and to rely on message aggregation to maximize the network bandwidth.

In our design, each worker (or helper) aggregates commands in large messages (buffers) and forwards them to a single communication server that in turn performs

[1]On Olympus we found multi-threading to have better performance with MVAPICH than with OpenMPI.

the MPI call. We consider this design effective if it maximizes the network bandwidth between two nodes. The optimal size of the aggregation buffers is a tradeoff between the bandwidth and the memory foot-print of using large buffers. We found a buffer size of 64KB to be a good compromise in our experiments with Olympus.

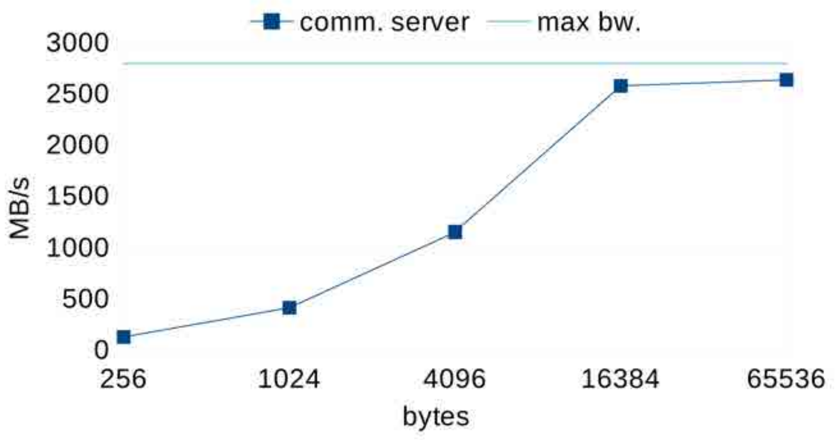


Figure 5.4: Bandwidth between two nodes using a single Communication Server and a single worker with varying message size.

Figure 5.4 shows the bandwidth reached between two nodes when using one worker and one communication server while varying the message size. The maximum bandwidth with this configuration is reached with 64KB messages, and is equal to 2630 MB/s slightly below the measured MPI network bandwidth of 2815.01 MB/s with the same message size. In the next section we describe how we implemented aggregation to coalesce commands into large buffers.

5.5.3 Aggregation

Data aggregation allows efficient exploitation of the available network bandwidth in presence of the fine grained data accesses typical of irregular applications. GMT accumulates commands directed towards the same destination nodes and sends them in bulk. These commands are then unpacked and executed at the destination node.

To increase the opportunity of aggregating network transfers, GMT uses *aggregation queues* to collect request or reply commands with the same destination from all the workers and helpers of a node. To this aim, GMT employs high-throughput, non-blocking concurrent aggregation queues. Nevertheless, the cost of concurrent accesses is too high if performed for every generated command. Hence, GMT implements a preaggregation phase. This phase initially collects commands in *command blocks*, local to each worker or helper, before inserting them into the shared aggregation queues.

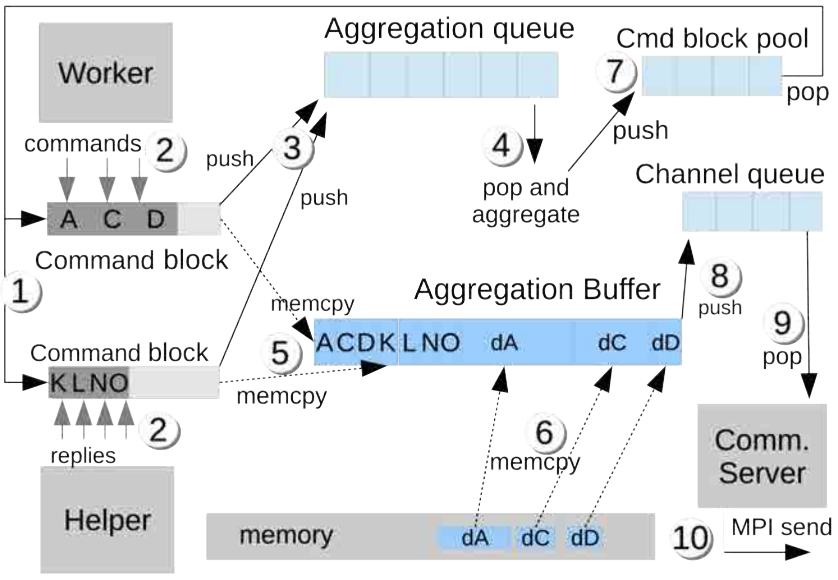


Figure 5.5: Aggregation mechanism

Figure 5.5 describes the aggregation mechanism in GMT. When a worker or a helper thread starts generating commands, it requests one of the pre-allocated command blocks from the command block pool (1). Command blocks are re-usable arrays containing several commands. They are pre-allocated and recycled for performance reasons. While a worker executes the application code, it generates commands of various types that are collected into the local command block (2). Helpers also generate commands when creating replies for incoming operations. In the example, a worker generates commands A,C,D and a helper generates commands K,L,N and O. Commands can be any type of remote request (*get*, *put*, *atomic* etc.). Waiting until the command block is full may increase too much the latency, for this reason workers or helpers push command blocks into the aggregation queue (3) when one of the following conditions is verified: i) the command block is full; ii) the command block has been waiting longer than a predetermined time interval (clock cycles). A command block is considered full when all the available entries are occupied with commands, or when the size in bytes of the commands, with the attached data, reaches the maximum size of the aggregation buffer to be sent.

Aggregation queues are shared among all the workers and helpers in the node, and there is one of them for each destination node. The aggregation consists in copying commands into an *aggregation buffer* that is sent over the network to the destination node. When a worker (or a helper) finds that the aggregation queue for a destination node has enough command blocks to fill an aggregation buffer (in terms of number of commands or in equivalent byte size), the actual aggregation starts. Aggregation can also start because the aggregation queue has been waiting longer than a predetermined time interval. When aggregation starts, the worker (or helper) pops command blocks from the aggregation queue to fill the aggregation buffer. GMT uses a fixed pool of aggregation buffers that are recycled to save memory space and eliminate allocation overhead. Multiple commands are copied from their command block into the aggregation buffer at once (5). For commands that require data movement (such as *gmt_put()* or the reply to a *gmt_get*), the data is also copied from the memory into the aggregation buffer. In the example in Figure 5.5, the data for the commands A, C, D is represented as dA, dC and dD respectively. After the copy, commands blocks are returned to the command block pool (7). The aggregation algorithm continues to push command blocks until an aggregation buffer is full. When this happens, the worker (or helper) pushes the aggregation buffer into a channel queue (8). Channel queues are high-throughput single-producer single-consumer queues that enable the communication between a worker (or helper) and the communication server.

The communication server continuously polls the channel queues, checking if new filled aggregation buffers are available. If so, the communication server pops a filled aggregation buffer and performs a non-blocking MPI send. It then returns the aggregation buffer into the pool of available aggregation buffers (not represented in the figure).

It is worth mentioning that GMT enables further tuning of design parameters such as local command block, queue and pool sizes, or time intervals, to optimize aggregation for different processor and network interconnect architectures.

5.5.4 Multithreading

Concurrency, through fine-grained software multithreading, allows tolerating the added latency for aggregating communication operations. We use the term *task* to identify a function pointer and an hardware execution context (stack, heap, hardware registers etc.) inside GMT, while we use the term specialized thread (or, simply, thread) to identify either a worker, a helper or the communication server. Each worker executes a set of GMT tasks. The worker switches among tasks' contexts every time it generates a blocking command that requires a remote memory operation. The task that generated

ctxt switches	**1 task**	**8 tasks**	**64 tasks**	**1024 tasks**
1	1816.00	1500.25	1536.81	1799.10
100	497.31	496.71	554.25	590.91
1000	517.14	494.56	545.00	579.13

Table 5.6: Latency (clock cycles) of a context switch when increasing the number of tasks and the number of context switches per task.

the command executes again only when the command itself completes (i.e., it gets a reply back from the remote node). In case of non-blocking commands, the task continues executing until it encounters a *gmt_wait_commands()* primitive.

GMT implements custom context switching primitives that avoids some of the lengthy operations (e.g., saving and restoring signal mask) performed by the standard *libc* context switching routines [78]. To evaluate the maximum network latency that is potentially tolerable, we measured the cost of context switching among two or more tasks. We performed an experiment that executes an increasing number of context switches among an increasing number of tasks. Each of the T tasks performs N context switches in a *for* loop. We run experiments where N is 1, 100 or 1000 context switches and T is 1, 8 64 or 1024 tasks. The latency of a single context switch is measured as the latency to perform N context switches in one of the tasks divided by N.

Table 5.6 shows the latency, in clock cycles, to execute a context switch with this experiment. When increasing the total number of context switches executed from 1 to 1000, we observe the effect of the caches that avoids retrieving the task context from memory. We also observe that the latency only slightly increases when increasing the number of tasks.

The optimal number of concurrent tasks per worker actually depends on the architecture (cache size) and on the workload (amount of work per task).

In GMT, the programmer typically generate tasks (except the task zero) by calling the *gmt_parFor()* construct. Figure 5.6 schematically shows how GMT executes a task. A node receives a message containing a *spawn* command (1) that a worker in a remote node generated when encountering a *gmt_parFor()* construct. The Communication Server passes the buffer containing the command to an helper, which parses it and executes the command (2). The helper then creates an *iteration block* (itb). The itb is a data structure that contains the function to execute, the arguments of the function itself, and the number of tasks that executes the same function. Each task represents a single iteration of the original *parFor*. This way of representing a set of tasks avoids

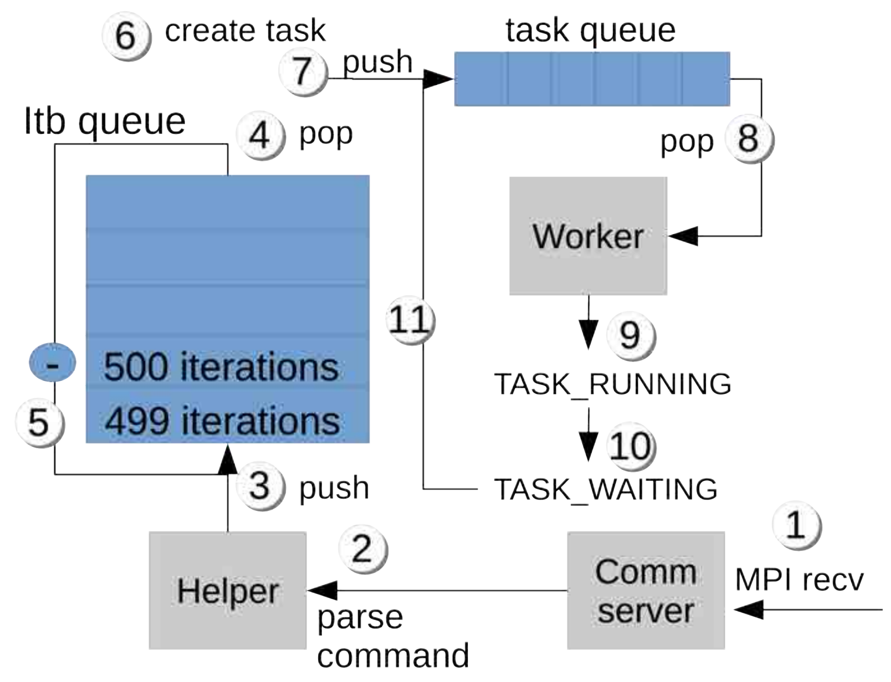


Figure 5.6: Fine grain multithreading in GMT.

the cost of creating a large number of function arguments and sending them over the network. In the following step, the helper pushes the iteration block is into the itb queue (3). Then, an idle worker pops the iteration block from the itb queue (5), decreases the counter of the iterations of t and pushes it back into the itb queue (6). The worker then creates t tasks (6) and pushes them into its private task queue (7). Then an idle worker pops a task from its task queue (8). If the worker can execute the task (i.e., all remote requests are completed), it restores the task's context and executes it (9) otherwise it pushes the task back into the task queue. The task contains user-level application code, which eventually calls one of the GMT primitives. In case the GMT primitive is a blocking remote request (e.g., *gmt_get()*), or an explicit wait (*gmt_waitCommands()*), and they are not completed, the task enters into a waiting state (10) and is reinserted into the task queue for future execution (11).

5.6 Experimental Evaluation

As introduced in section 5.5, we evaluated GMT on PNNL's Olympus supercomputer. GMT can adapt to other systems by tuning configuration parameters defined at installation time. For this work, we empirically optimized the parameters GMT for the

Olympus system. Table 5.7 presents the configuration used in our benchmarks.

Parameter	**Configuration**	
NUM_WORKERS	15	
NUM_HELPERS	15	
NUM_BUF_PER_CHANNEL	64	
SIZE_BUFFERS	65536	
MAX_NUM_TASKS_PER_WORKER	1024	

Table 5.7: GMT configuration parameters for Olympus.

Follows a brief explanation of the selection procedure for each parameter:

- *NUM_WORKERS* determines the number of worker threads per node . Because we assume that each thread is mapped to a core, and one core is taken by the communication server, they depend on the number of cores per node ($NUM\_WORKERS = (CORES - 1)/2$).
- *NUM_HELPERS* determines the number of helper threads. Same as NUM_WORKERS.
- *NUM_BUF_PER_CHANNEL* determines how many buffers are allocated to each communication channel. The number of buffers per channel allows the local worker (or helper) to continue sending network buffers even when the remote node is not consuming them. A safe value for this parameter is equal to the total number of nodes, but in case of network congestion a larger value can be used.
- *SIZE_BUFFERS* determines the size of the aggregation buffer. This depends on the network characteristics. Using Olympus we set this parameter to 64KB because there is the smaller message size that maximizes the throughput (see Figure 5.4).
- *MAX_NUM_TASKS_PER_WORKER* determines the maximum number of tasks concurrently executed by each worker. The optimal number of concurrent tasks should be large enough to overlap the latency of the network communication, but not too large to incur in too many cache misses during the context switch. We found the value of 1024 for our architecture trough empirical experimentation with GMT micro-benchmarks (see Table 5.6).

Initially, we analyze the peak communication performance of the full GMT infrastructure, focusing on the combined effects of aggregation, for optimizing bandwidth utilization, and of fine-grained software multithreading, for latency tolerance. We then discuss the performance of three irregular application kernels running on GMT: Breadth First Search (BFS), Graph Random Walk (GRW) and Concurrent Hash Map (CHM). We compare the GMT implementations of these kernels to corresponding implementations using MPI, UPC, the Cray XMT and OpenMP. We execute the GMT and UPC versions on the same cluster, while for the OpenMP version we use a 48-core system with 4 AMD Opteron 6176SE processors ("Magny-Cours" at 2.3 GHz) and 256 GB of DDR3 RAM at 1333 MHz. We compare these substantially different platforms by employing throughput metrics. We maintain consistent input sizes for each comparison. We select the experiments to describe specific aspects and behaviors of GMT, rather than providing an exhaustive comparison of all the implementations and the platforms.

5.6.1 Micro-benchmarks

In this section we characterize the peak communication performance of GMT. The aim of this characterization is to quantify the effects of the aggregation when performing a large number of basic GMT remote operations. When GMT executes a series of fine-grain put operations, we expect to observe a considerable performance improvement in bandwidth utilization with respect to sending MPI messages of the same size, because of aggregation. Furthermore, increasing the number of concurrent tasks increases the likelihood of generating communication operations. Thus, we expect that aggregate bandwidth increases with the number of concurrent tasks in the node.

Figure 5.7 shows how transfer rates between two nodes behave when increasing the number of tasks per node in GMT. Every task executes 4096 blocking put operations. All the experiments use 15 workers, but we increase the number of tasks for the node. The graph plots message sizes from 8B to 128 bytes. We verify that increasing the concurrency in the node increases the transfer rates, because there is a higher number of messages that GMT can aggregate. With 1024 tasks, puts of 8 bytes reach a bandwidth of 8.55 MB/s. With 15360 tasks, the bandwidth increases to 72.48 MB/s, a factor of 8.4. Larger messages provide higher bandwidth, because they reduce the network overhead. With messages of 128 bytes and 15360 tasks, GMT reaches almost 1 GB/s, while the best MPI implementation reaches 72.26 MB/s (using 32 processes). At these message sizes, with blocking operations, the task switching time also becomes a factor. In fact, a node should be able to generate as many network references as possible to saturate the effective network bandwidth for small messages. When concurrent tasks emit communication operations in parallel, they increase the injection rate. However, if the task switching time is too high, there is an added latency between an injected

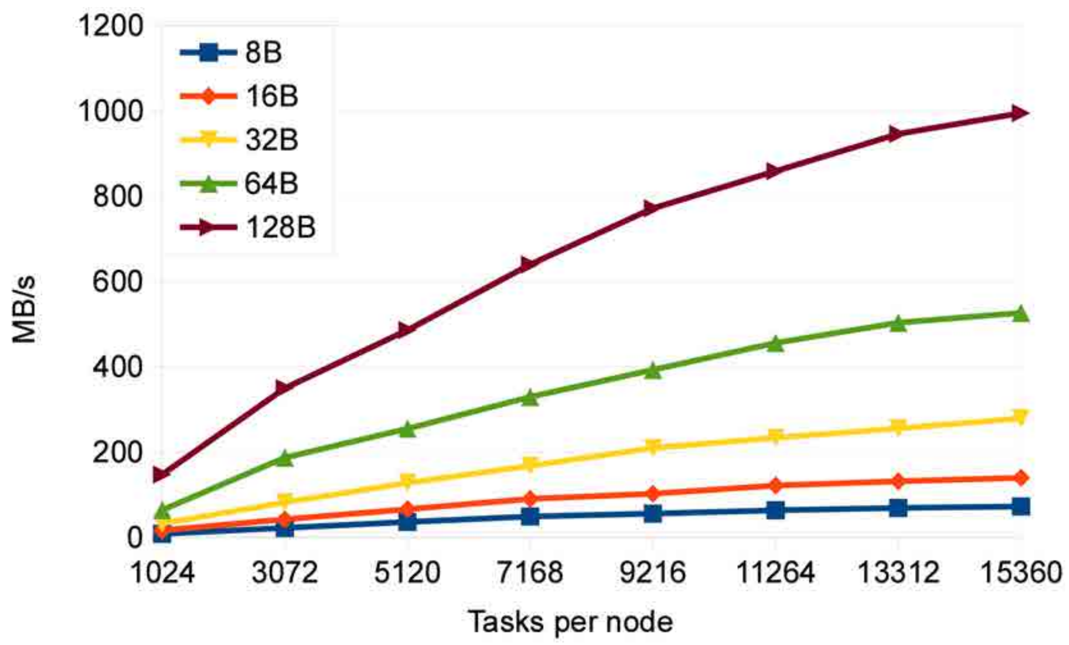


Figure 5.7: Transfer rates of put operations between 2 nodes while increasing concurrency. Multiple lines show the transfer rate with message sizes from 8 bytes to 128 bytes.

network operation and another which does not allow maximizing network utilization, even considering the overheads for packet headers.

With more destination nodes, the probability of aggregating enough data to fill a buffer for a specific remote node decreases. To verify the behavior of aggregation when communicating with a higher number of nodes, we executed the same experiment on 128 nodes.

Figure 5.8 shows the results. If we compare it to the previous figure, we observe a slight degradation in performance. However, aggregation is still very effective with respect to MPI send operations of the same size. For instance, GMT with messages of 16 bytes over 128 nodes reaches a bandwidth of 139.78 MB/s, versus the 9.63 MB/s of the MPI send operation (using 32 processes).

5.6.2 BFS

BFS is one of the most common graph search kernels, and a building block for many graph analysis applications. As a matter of fact, the BFS is part of the Graph500 [81] benchmark suite and a de-facto benchmark for irregular applications. All the implementations exploit parallelism on the vertex queue while progressing through the various

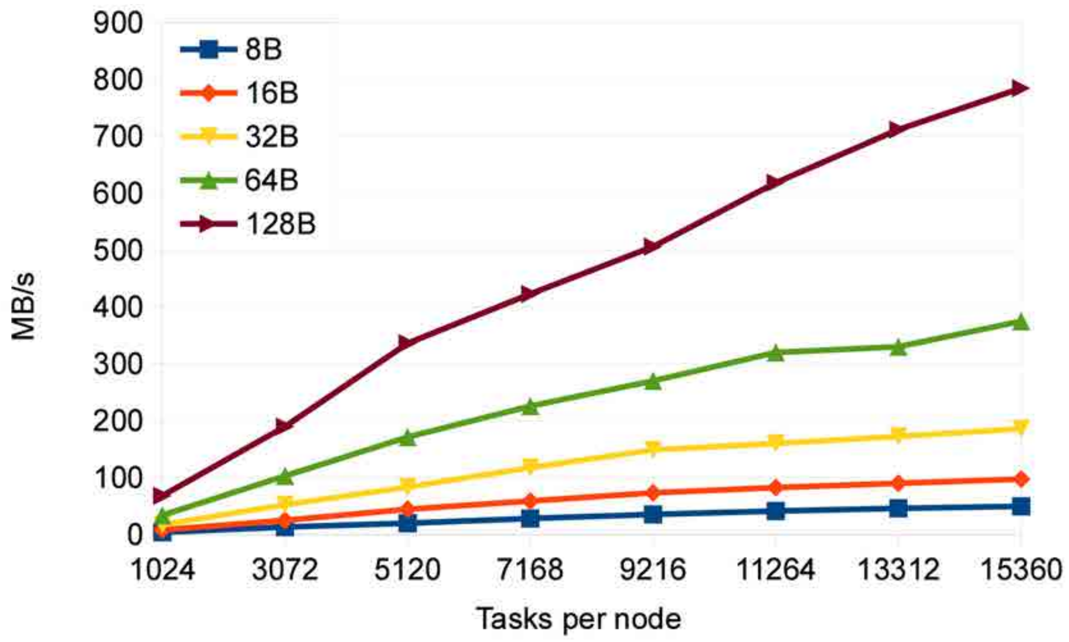


Figure 5.8: Transfer rates of put operations among 128 nodes (one to all) while increasing concurrency. This is the transfer rate of the outgoing messages from the source node. Multiple lines show the transfer rate with message sizes from 8 bytes to 128 bytes.

levels of exploration. The code for GMT, Cray XMT and OpenMP is essentially identical, except that GMT primitives are substituted with Cray XMT proprietary primitives and OpenMP compiler pragmas. The code for UPC instead, uses several optimizations such as caching the exploration map, aggregating communication at the application code level and using asynchronous gets for the aggregated transfers. Table 5.8 shows the number of lines of code for GMT, UPC and MPI implementations of the BFS.

implementation	**Lines of code**
GMT	88
UPC	715
MPI	103

Table 5.8: Lines of code - Breadth First Search implementations

Figure 5.9 shows the weak scaling of the GMT implementation, measured in million of traversed edges per second. The implementation is the one presented in Figure 5.2, which performs single-word memory accesses on the global graph structure. For this experiment we used randomly generated graphs, increasing the size of the graph of 1 million vertices for each node added. Each vertex has at most 4000 edges connecting

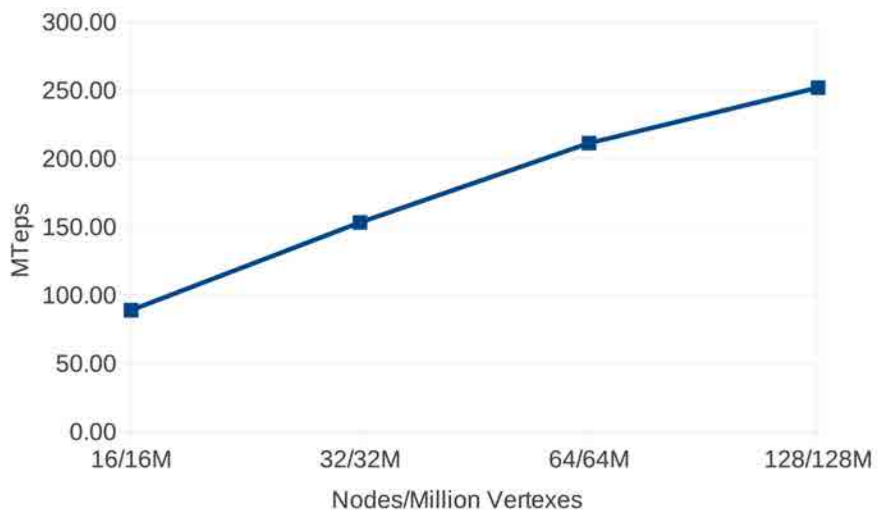


Figure 5.9: Million traversed edges per second for the GMT implementation of the BFS (weak scaling)

to random vertices in the graph. Therefore, the largest graph on the 128 nodes configuration has 128 million vertices and 258 billion edges, for a total memory footprint of ≈ 2 TB.

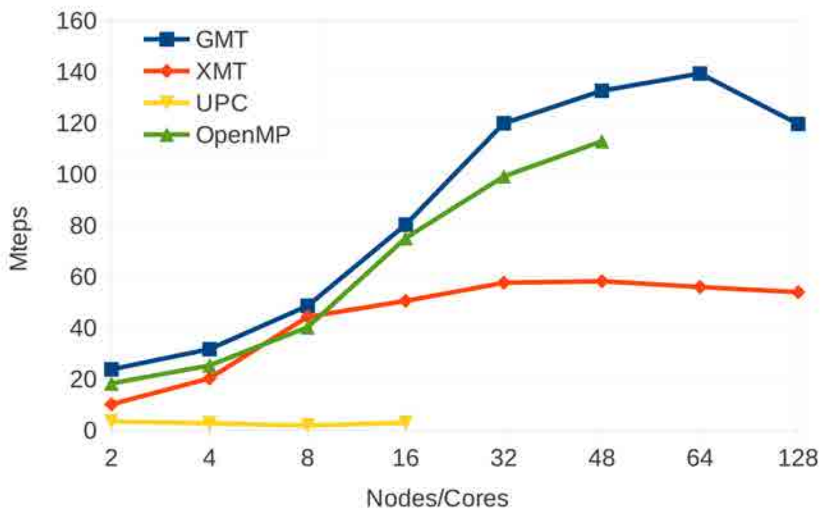


Figure 5.10: Million traversed edges per second for the BFS implementation on GMT, UPC, Cray XMT, OpenMP (strong scaling). The horizontal scale for GMT, XMT and UPC represents cluster-nodes while for OpenMP represents cores

Figure 5.10 shows the strong scaling of the GMT implementation, comparing it to the equivalent queue-based implementations for UPC, the Cray XMT and OpenMP.

For these experiments, we used a random graph of 10 million vertices and 2.5 billion edges, due to the maximum memory capacities of the platforms (256 GB on the AMD shared memory system and 1 TB on the Cray XMT). An interesting consideration is the amount of parallelism necessary to fully utilize the various platforms. While OpenMP needs a thread per core and the Cray XMT needs 128 threads per processor, GMT requires 1024 user tasks per worker. With 128 nodes and 15 workers per node, GMT needs 2 million tasks to fully utilize the system. Indeed, GMT's performance starts to decrease above 64 nodes, because the available parallelism in the application is not enough. As the graph in Figure 5.10 shows, the BFS implementation for GMT outperforms the other implementations. Even for a relatively small graph, GMT running on a cluster outperforms OpenMP running on a single, large shared memory node. This result is of particular interest, although not obvious, because each node of the cluster running GMT is a SMP node. In fact, the performance of irregular algorithms is mainly limited by the bandwidth for accessing data in unpredictable locations. For a SMP workstation, the bottleneck is thus the memory bandwidth. For a distributed memory system, instead, the main bottleneck is the network interconnection, which usually is significantly slower than the memory subsystems, especially with fine-grain data transfers. Thus, GMT effectively optimizes the network bandwidth utilization, meeting its objective of increasing not only the size of the tractable problems, but also the processing performance as more nodes are added. Finally, we underline that the programming effort for GMT is essentially identical to OpenMP and the Cray XMT, while the UPC version was significantly more challenging to implement and optimize.

5.6.3 Graph Random Walk

Graph Random Walk (GRW) randomly traverses a graph with the purpose of collecting vertex/edge information or of understanding graph properties. Many application areas, such as artificial intelligence, brain research and game theory, exploit the GRW kernel in many algorithms. In a GRW, each task starts from a *source* node, chooses randomly a neighbor to visit, and continues the walk until it has visited L connected nodes. Our code, given a connected graph of V vertices and E edges, assigns $V/2$ vertices as *source* nodes to $V/2$ parallel tasks. Each task performs a walk of length L. Implementing the GRW in GMT is fairly simple: (i) *gmt_parfor()* spawns $V/2$ tasks; (ii) each task performs a random walk of length L, accessing the graph with GMT primitives. We compare the GMT implementation to a state-of-the-art MPI implementation employed in fast matching algorithms [40]. This approach, rather than making a process retrieve non-local data, delegates the completion of a walk to the process that locally owns the data. The algorithm employs P processes, divided as one master process and $P-1$ slave processes. Given a graph with V vertices and E edges, the algorithm performs the following steps: (1) the master process initializes and distributes V/P vertices to all P processes (including itself); (2) each process starts $V/(P*2)$ walks ($V/2$ walks in total for the whole system) of length L from each one of its assigned vertices; (3) if a vertex v is not owned by the current process, it delegates the process owning v to continue the walk; (4) when a process completes all its local walks, it communicates to the master the number of completed walks (i.e., walks that traversed L nodes) and waits for walks to continue from other processes (in case there are new walks, it restarts from step 3); (5) when all the $V/2$ walks are completed, the master sends the quit command to the slave processes; Furthermore, the MPI algorithm exploits message aggregation to reduce fine-grain communication. Whenever a process requires the delegation of walks to other processes, it buffers all the requests for each process and sends them out at once only after completing the local walks (i.e., end of step 4). To empirically quantify the complexity of the two implementations, we measured the source lines of code for the GMT and MPI implementations. Table 5.9 shows the number of lines of code of the MPI and GMT implementations of the graph random walks.

implementation	**Lines of code**
GMT	164
MPI	503

Table 5.9: Lines of code - Graph Random Walks implementations

Figure 5.11 shows (in logarithmic scale) the performance of the GRW, measured in million of traversed edges per second (MTEPS). The experiments use a randomly generated graph of one million vertices per-node (weak scaling) with an average of 4000 edges per vertex. The figure shows that GMT is one or more orders of magnitude faster than the MPI implementation.

5.6.4 Concurrent Hash Map Access

Concurrent Hash Map Access (CHMA) is a synthetic benchmark where multiple concurrent activities access a hash map to check the presence of a hashed element (e.g., a string or a signature). If the element is found, it is modified according to a predetermined rule and stored back into the hash map. The behavior of this kernel is typical of streaming applications such as virus scanning, spam filters, natural language

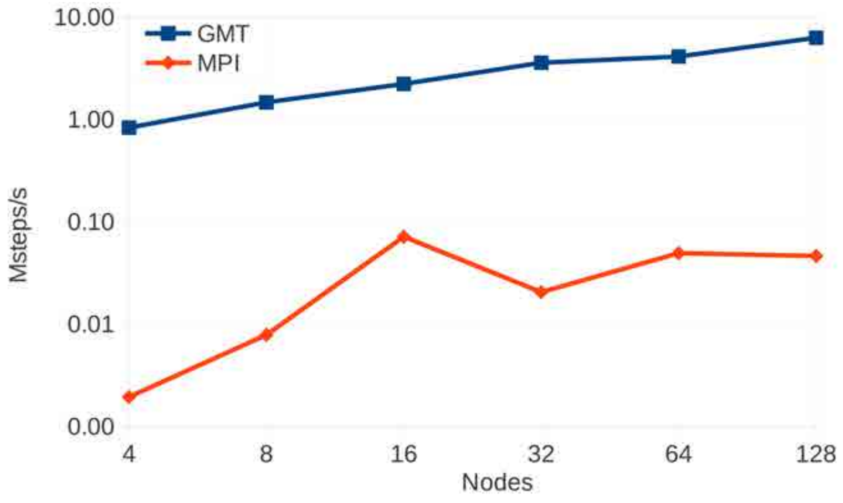


Figure 5.11: Millions of steps per second for the random walk implementation on GMT and MPI (weak scaling)

processing, and of information retrieval applications that need to store, filter and manipulate large amounts of streaming data. In our experiments, we used a pool of 100 million strings with at most 20 characters each to populate a hash map of 10 million entries. After the initialization, W concurrent tasks perform the following operations for 'L' steps: (1) start from a given input string; (2) find if it is present in the hash map; (3) if it is present, perform a string reverse operation; (4) hash the new string and store it back in the hash map; (3) if it is not present, get a new input string. We compare both an MPI and a GMT implementation. In the MPI implementation, each MPI rank is responsible for a portion of the hash map. Only the process that owns the related portion of the hash map checks and inserts the strings. However, if the current process does not own the hashed string, it sends the string to its owner. Small MPI messages are very frequent, because a process cannot proceed with a new string until it has finished manipulating the previous one. It is possible to implement partial caching with remote bulk updates, but it requires employing expensive checks and invalidation mechanisms. On the other hand, the GMT implementation is straight forward: the *gmt_parfor()* construct spawns W tasks, each task independently performs get/put and atomic compare and swap operations on the hash map for L steps. As for the other two kernels, the MPI solution for CHMA was significantly more complex and difficult to implement than the GMT code. Table 5.10 shows the number of lines of code of the MPI and GMT implementations of the concurrent hash map access.

Figures 5.12 and 5.13 respectively show the throughput, in million of strings hashed

implementation	Lines of code
GMT	177
MPI	698

Table 5.10: Lines of code - Concurrent Hash Map Access implementations

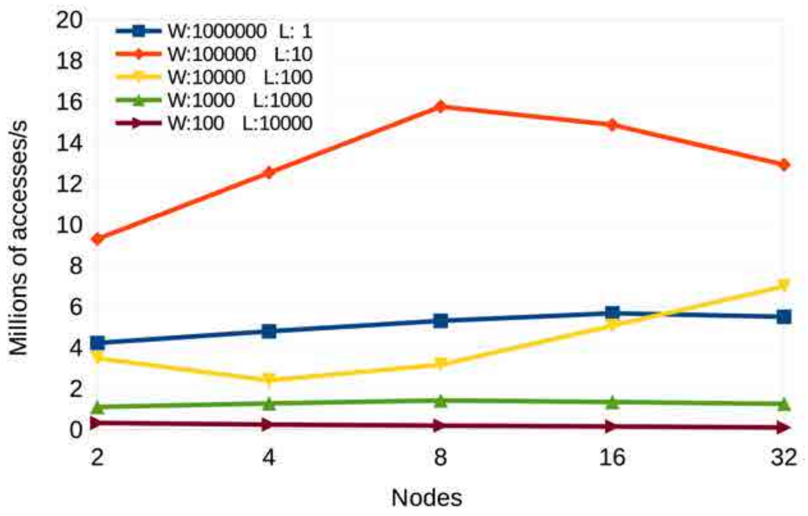


Figure 5.12: Number of strings hashed and inserted per second (Millions of accesses/s) for the GMT implementation of the Concurrent Hash Map Access benchmark. In the legend, W refers to the number of tasks and L to the number of accesses performed by each task.

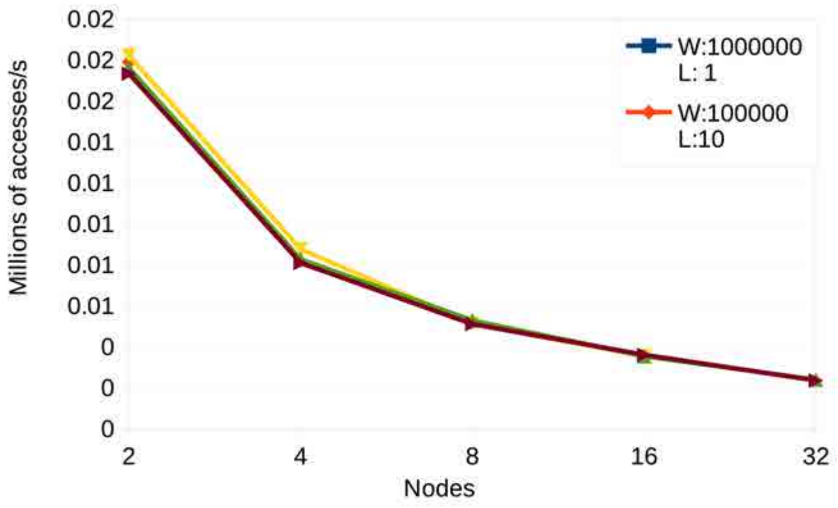


Figure 5.13: Number of strings hashed and inserted per second (Millions of accesses/s) for the MPI implementation of the Concurrent Hash Map Access benchmark. In the legend, W refers to the number of processes and L to the number of accesses performed by each process.

and inserted per second (Millions of accesses/s) of the GMT and the MPI implementations, while increasing the number of cluster nodes and varying the number of tasks (or processes) that concurrently accesses the hash map (W) and the number of steps (L) performed by each tasks (or process). The performance between the GMT and the MPI implementations differs by two or more orders of magnitude, because of the fine grained communication involved in the kernel.

5.7 Conclusions

In this chapter we presented **GMT**, a Global Memory and Threading library that enables efficient execution of irregular applications on commodity clusters. GMT integrates a PGAS data substrate with simple loop parallelism. It provides a simple interface for designing applications with large, irregular data structures, without requiring data partitioning. It allows expressing and extracting the large amounts of fine grained, dynamic parallelism present in irregular applications through simple parallel loop constructs. GMT's architecture employs specialized threads to realize its functionality: workers, for executing the applications, helpers, to support data communication, and a communication server, to perform data transfers. GMT is built around the concepts of lightweight user level multithreading and data aggregation to reduce the impact of fine grained, unpredictable data accesses typical of irregular applications. GMT tolerates network communication latencies by switching thousands of tasks on each available worker thread. GMT implements multi-level aggregation to maximize network bandwidth utilization with small messages. This is in contrast with current PGAS infrastructures, which exploit a SPMD control model and are optimized for regular data access and block transfers. GMT is easily portable across different systems: the only pre-requisites are MPI support and, currently, x86 processors for the task switching routines. GMT aims at providing a solution to scale irregular applications in performance and size by adding more nodes to a cluster. We characterized the communication performance of GMT, and compared it to UPC and MPI hand-optimized code, as well as custom machines designed for irregular applications, on a set of typical large scale, irregular application kernels. We demonstrated speed ups of orders of magnitude compared to other solutions for commodity clusters, and performance comparable to custom machines.

Part IV

User-level Scalability Exploiting Hardware Features

Chapter 6

Exploiting Hardware Thread Priorities on Multi-threaded Architectures

The previous part of the thesis described the design and implementation of a specialized runtime system for large-scale high performance applications. This part of the thesis focuses on user-level hardware features that can be exploited into a runtime system to improve applications performance. This chapter describes the performance benefits of exploiting SMT hardware-thread priorities of IBM POWER5 and POWER6 processors. This work is a study to evaluate hardware optimizations that could be integrated into an adaptive runtime system for a specialized class of applications.

6.1 Summary

The limitations in exploiting instruction-level parallelism (ILP) has motivated thread-level parallelism (TLP) as a common strategy to improve processor performance. There are several TLP paradigms which offer different benefits as they exploit TLP in different ways. For example, Simultaneous multi-threading (SMT) reduces fragmentation in on-chip resources. In addition to SMT, Chip-Multiprocessing (CMP) is also effective in exploiting TLP with limited transistor and power budget. This motivates processors vendors to combine both TLP paradigms in their processors. For instance Intel i7 as well as IBM POWER5 and POWER6 combine SMT and CMP.

Because SMT processors share most of the core resources among threads, some of them implement mechanisms to better partition the shared resources. For instance, the

2-way SMT processors POWER5 and POWER6 improve the usage of resources across threads with mechanisms in hardware [97][105] that suspend a thread from consuming more resources when it stalls for a long-latency operation. One of the interesting new features which enables better resource balancing is that POWER5 and POWER6 allow software to control the instruction decode rate of each thread in a core by eight priority levels, from 0 to 7. The higher the priority difference between the two threads in each core, the higher the difference of decode cycles, and hence, the difference of hardware resources received by the two threads.[1]

The Operating System (OS) can provide a user interface to change thread priorities such that software can control the speed at which each hardware thread run with respect to the other hardware thread in a core. The default priority configuration (i.e., both hardware threads having priority 4) is designed to guarantee fair hardware resource allocation between hardware threads. From a software point of view, the main motivation to override the default configuration is to address instances where non uniform hardware resource allocation is desirable. Several examples can be enumerated such as virtualization in SMT, OS idle thread, thread waiting on a spin-lock, latency-sensitive threads, software determined non-uniform balance and power management [57][97][149]. In some cases, software-controlled thread priorities can also improve instruction throughput or parallel applications execution time [33][34], by optimizing hardware resource allocation. Although software-controlled thread priorities have a considerable potential, lack of quantitative studies limits their use in real world applications. In this work, we provide a quantitative study of the POWER5 and POWER6 prioritization mechanism. We show that the effect of thread prioritization depends on the characteristics of the two threads running simultaneously in a core. We also show that thread priorities have different effects on applications in POWER5 and POWER6. We analyze the major processor characteristics that lead to this different behavior. In particular, although both processors are dual-core and each core is two-way SMT, their internal architectures are different. While POWER5 has out-of-order cores with many hardware shared resources, POWER6 follows a high-frequency design optimized for performance, leading to a mostly in-order design in which fewer resources are shared between threads. Finally, we show the benefits of software-controlled thread priorities in real-world applications including parallel applications and multi-programmed environments.

[1]Note that software-controlled hardware priorities are independent of the operating systems concept of process or task prioritization. In fact, task priorities are used to prioritize scheduling of running tasks among CPU's and, therefore, are a pure software concept.

We define *SMT thread malleability* (or simply *malleability*) as the ratio between the performance of a thread with a given priority configuration and its performance with default priority configuration. To characterize POWER5 and POWER6 thread prioritization mechanism we developed a set of micro-benchmarks that stress specific hardware resources such as data cache, issue queues, and memory bus. Moreover, we measure the malleability of real workloads, represented by some of the SPEC CPU2006 benchmarks [86]. Also, we develop a Linux kernel patch that provides an interface to the user to set all possible priorities available in kernel mode. Without a kernel patch, only three of the eight priorities are available to the user.

6.2 Experimental environment

Tables 6.1 and 6.2 show the experimental environment used in this chapter. The tables include the hardware systems, the communication libraries, and the benchmarks used for the experiments.

CPU	IBM POWER5 processor at 1.65 Ghz (1 socket - 2 cores per socket - 2 SMT threads per core)
Memory	8 Gigabyte of DDR RAM
Network	Ethernet (not used)
System size	single node
Operating System	Linux
Communication Library	MPI
Benchmarks	SPEC CPU2006, NAS Parallel Benchmarks

Table 6.1: Experimental environment - IBM Open Power 710 system

CPU	IBM POWER6 processor at 4.0 Ghz (2 sockets - 2 cores per socket - 2 SMT threads per core - no L3 cache)
Memory	16 Gigabyte of DDR2 RAM
Network	Ethernet (not used)
System size	single node
Operating System	Linux 2.6.23
Communication Library	MPI
Benchmarks	SPEC CPU2006, NAS Parallel Benchmarks

Table 6.2: Experimental environment - IBM JS22 system

6.3 Related Work

Some previous studies focus on ensuring QoS in SMT architectures. Cazorla et al. introduce a mechanism to force predictable performance in SMT architectures [42]. They manage to run time-critical jobs at a given percentage of their maximum IPC. To attain this goal, they need to control all shared resources of the SMT architecture.

Regarding CMP architectures, Rafique et al. propose to manage shared caches with a hardware cache quota enforcement mechanism and an interface between the architecture and the OS to let the latter decide quotas[139]. Nesbit et al. introduce Virtual Private Caches (VPC) [127], which consists of an arbiter to control cache bandwidth and a capacity manager to control cache storage. They show how the arbiter allows meeting QoS performance objectives or fairness. A similar framework is presented by Iyer et al. [92], where resource management policies are guided by thread priorities. Individual applications can specify their own QoS target (e.g. IPC, miss rate, cache space) and the hardware dynamically adjusts cache partition sizes to meet their QoS targets. An extension of this work with an admission mechanism to accept jobs is presented in [82].

Also, previous works show that SMT performance heavily depends on the nature of the concurrently running applications [57][151]. Tuck et al. analyze the performance of a real SMT processor [157], concluding that SMT architectures provide an average speedup over single-thread architectures of about 20% and that, even if the processor is designed to isolate threads, performance is still affected by resource conflicts.

Other works propose the use of hardware thread priorities to control thread execution in SMT processors. Many of these proposals implement fetch policies to maximize throughput and fairness by reducing the priority, stalling, or flushing threads that experience long latency [43, 158]. Boneti et al. analyze the effect of hardware priorities on POWER5 [33], and use hardware priorities to balance resources in SMT processors [35] and to implement dynamic scheduling for HPC [34].

In this context, the concept of Fair CPU utilization accounting for CMP and SMT processors, introduced by Luque et al. [112][113] and by Eyerman and Eeckhout [65], can be used to improve the efficiency of thread prioritization mechanisms.

Let us assume a workload composed by several tasks ($Ta,\ Tb,\ \ldots\ ,\ Tn$) running in an n-core multicore or n-way SMT processor. The mechanisms proposed [65][112][113] provide an estimation of the execution time that each of these tasks would have if it runs in isolation ($Ti\_isolation$). By measuring the difference in execution time between the execution time in CMP/SMT and the execution time in isolation ($Ti\_cmp/Ti\_isolation$

or *Ti_smt/Ti_isolation*), we can determine the slowdown each task is suffering in CMP/SMT. The slowdown (or the relative speed) could be used to guide the SMT prioritization mechanism (or any other prioritization mechanism for multicores such as the one described by Moreto et al. [120]) to ensure Quality of Service, that is, to ensure that tasks do not suffer a performance degradation greater than a pre-established threshold.

To our knowledge, this part of a first extensive study that quantify the effect of hardware-thread priorities on two SMT processors with substantially different microarchitecture, such as POWER5 and POWER6.

6.4 POWER5 and POWER6 Microarchitecture

This section provides a brief description of POWER5 and POWER6 microarchitecture and of the features that are relevant to SMT and thread priorities. A detailed description of the processors can be found in the works of Sinharoy et al. [105] and Le et al. [149].

6.4.1 POWER5 and POWER6 Core Microarchitecture

Figure 6.1 shows a high-level diagram of POWER5 and POWER6 processors. Both processors have two cores and each core supports 2-thread SMT. In both processors, each core has its own L1 data and instruction cache. In POWER5, L2 cache is shared among cores whereas in POWER6 each core has its own L2 cache. In both processors, the off-chip L3 cache is shared. POWER6 microprocessor has a ultra-high frequency core and represents a significant change from POWER5 design. Register renaming and massive out-of-order execution as implemented in POWER5 are not employed in POWER6. However, POWER6 implements limited out-of-order execution for floating point instructions [105].

6.4.2 Simultaneous Multi-Threading

POWER5 has separate instruction buffers for each thread. Based on thread prioritization, up to five instructions are selected from one of the instruction buffers and a group is formed. Instructions in a group are all from the same thread. POWER6 core implements an independent dispatch pipe with a dedicated instruction buffer and decode logic for each thread. At the dispatch stage, each group of up to five instructions per thread is formed independently. Later, these groups are merged into a dispatch group of up to seven instructions to be sent to the execution units. Several other features

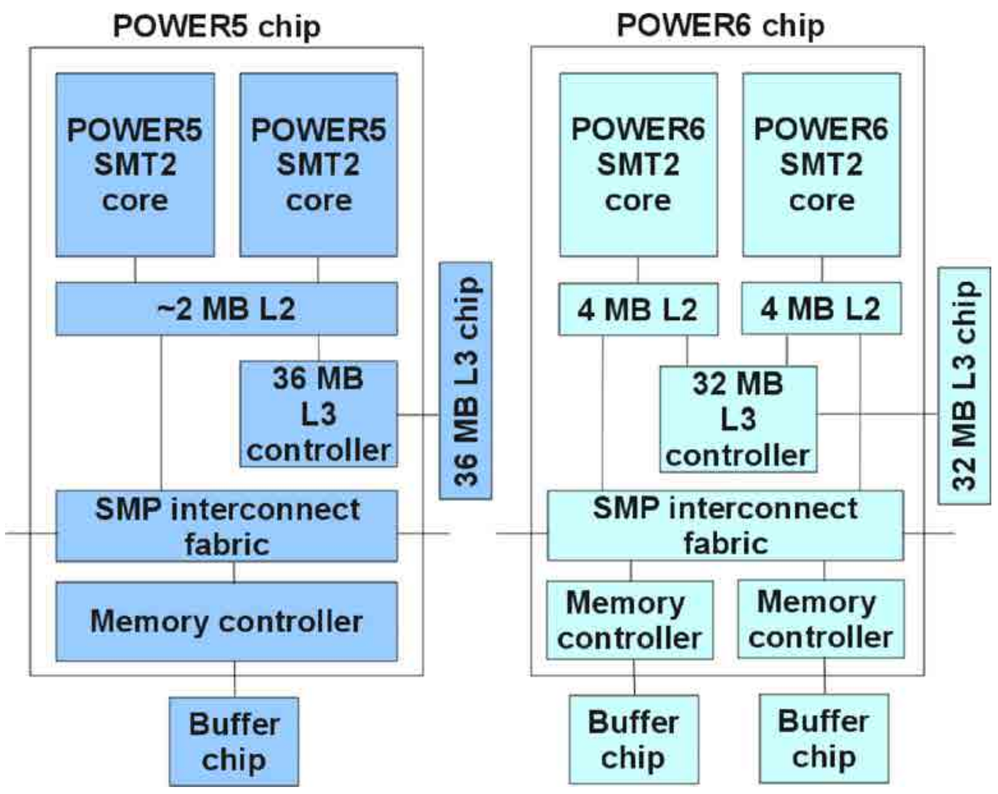


Figure 6.1: POWER5 and POWER6 architecture

have been implemented in POWER6 to improve SMT performance. For instance, the L1 I-cache and D-cache size and associativity have been increased from the POWER5 design. POWER6 core has dedicated completion tables (GCT) per thread to allow more outstanding instructions [105].

Both processors deploy two levels of resource control among threads through dynamic resource balancing in hardware and through thread prioritization in software. POWER5 and POWER6 dynamic hardware resource-balancing mechanisms monitor processor resources to determine whether one thread is potentially blocking the other thread execution. Under that condition, the progress of the offending thread is throttled back allowing the sibling thread to progress (*automatic throttling mechanism*). For example, POWER5 considers that there is an unbalanced use of resources when a thread reaches a threshold of L2 cache misses or TLB misses, or when a thread uses too many GCT (reorder buffer) entries [149].

6.4.3 Software-controlled Hardware Thread Priorities

In POWER5 and POWER6, software-controlled priorities range from 0 to 7, where 0 means the thread is switched off and 7 means the thread is running in Single Thread (ST) mode (i.e., the other thread is off).

Using priority 1 for both threads has the effect of executing the threads in low-power mode. In addition, the execution of one thread with priority 1 while the other has a priority > 1 causes the former to use only hardware resources leftover by the latter.

The enforcement of software-controlled priorities is carried in the decode stage. In general, a higher priority translates into a higher number of decode cycles. In POWER5, assuming a primary thread and a secondary thread[1] with priorities P and Q (where $P > 1$ and $Q > 1$) , decode cycles are allocated as follows:

1. compute R:
$$R = 2^{|P-Q|+1}$$

2. decode cycle rates:
$$r_{high} = (R-1)/R$$
$$r_{low} = 1/R$$

Where r_{high} is the decode cycle rate of the thread with higher priority and r_{low} is the decode cycle rate of the thread with lower priority. The thread with higher priority receives $R-1$ every R decode cycles, while the thread with lower priority receives 1 every R decode cycles. For instance, assuming that the primary thread has priority 6 and the secondary thread has priority 2, R would be 32, so the core decodes 31 times from the primary thread ($r_{high} = 31/32$) and once from the secondary thread ($r_{low} = 1/32$). Hence, the performance of the process running as primary thread increases to the detriment of the one running as secondary thread.

In the special case when threads have the same priority, R would be 2, and each thread alternately receives one slot ($r_{high} = r_{low} = 1/2$).

The previous formula is available for POWER5, while for POWER6 we assume that

[1] Primary thread and secondary thread are just naming conventions because the two hardware-threads are perfectly symmetrical.

the decode cycle rate is a monotonic function of the priority difference:

$$r_{high} = f_{high}(|P - Q|)$$
$$r_{low} = f_{low}(|P - Q|)$$

Table 6.3 shows priority values and levels, required privilege levels, and instructions used to set priorities. Supervisor or OS can set six of the eight priorities ranging from 1 to 6, while user software can only set priority 2, 3, and 4. The hypervisor can use the whole range of priorities.

Table 6.3: Thread priorities in POWER5 and POWER6

priority	**priority level**	**privilege level**	**instruction**
0	Thread off	Hypervisor	-
1	Very low	Supervisor	`or 31,31,31`
2	Low	User/Supervisor	`or 1,1,1`
3	Medium-Low	User/Supervisor	`or 6,6,6`
4	Medium	User/Supervisor	`or 2,2,2`
5	Medium-high	Supervisor	`or 5,5,5`
6	High	Supervisor	`or 3,3,3`
7	Very high	Hypervisor	`or 7,7,7`

Priorities can be set by issuing a pseudo *or* instruction in the form of *or X,X,X*, where X is a specific register number [73][91]. This operation only changes the thread priority and performs no other operation. In case it is not supported (i.e., running on previous POWER processors), or in case of insufficient privileges, the instruction is simply treated as a *nop*.

6.5 Experimental Setup

In order to explore the capabilities of the software-controlled priority mechanism in the POWER5 and POWER6 processors, we perform a detailed set of experiments. Our approach consists in measuring the performance of micro-benchmarks running in SMT mode as the priority of each thread is increased or reduced.

The performance of a process in an SMT processor are conditioned by the programs running simultaneously on the other hardware-thread, and by their phase. Evaluating all the possible programs and all their phase combinations is infeasible. Moreover, the evaluation of a real system, with several layers of running software, OS interferences and all the asynchronous services, becomes even more difficult.

For this reason, we use a set of micro-benchmarks that stress particular processor characteristics. While this scenario is not typical with real applications, it is a systematical way to understand the hardware priority mechanism. This methodology in fact, provides a uniform characterization based on specific program characteristics that can be mapped into real applications.

To verify the effects of hardware priorities on real applications, we measure the malleability of a subset of SPEC CPU2006 benchmarks with different priority configuration. To ensure that all the benchmarks are fairly represented in the final results, we use the FAME (FAirly MEasuring Multithreaded Architectures) methodology [162][163] which requires running in SMT mode the same benchmark pair for multiple times until both benchmarks are equally represented in the total execution time.

Running all pair-wise combinations of SPEC CPU2006 benchmarks and all priority combinations with FAME methodology would take too much time to complete.[1] In order to reduce experimentation time, we choose a subset of SPEC CPU2006 as follows: 1) we choose benchmarks such that the spectrum of performance, memory and execution unit characteristics are fairly represented in the subset, 2) following Snavely et al. [151] recommendation on symbiotic OS scheduling, we pair high-IPC (CPU intensive) benchmarks with low-IPC (memory-bound) benchmarks, in order to provide efficient utilization of the SMT core.

High-IPC benchmarks are *bzip*, *calculix*, *cactusADM* and *h264ref*. Low-IPC benchmarks are *mcf*, *omnetpp* and *milc* . The resulting combination represents mixes of high-IPC and low-IPC benchmarks as well as integer and floating point benchmarks.

6.5.1 Experimental environment

The results presented are obtained by compiling the benchmarks with *gcc* version 4.1.2 20070115 (SUSE Linux), Linux kernel version 2.6.23, *libpfm-3.8*, and *mpich2-1.0.8*. We executed the experiments on an Open Power 710 (Op710) and on a JS22 IBM server, with the same executable. It is worth noting that the Op710 POWER5 processor is equipped with a third-level (L3) cache while the JS22 POWER6 processor we use does not have the third-level cache.

[1]Running all pair-wise combinations of the 26 SPEC CPU2006 benchmarks with all combinations of the six priorities amounts to 19,500 possibilities; With an estimated average running time of two-hour-per-possibility using FAME methodology, a non-subset experiment would take a humbling four and a half years to complete on a single machine.

6.5.2 The Linux kernel modification

Some of the priority levels are not available in user mode (Section 6.4.3). In fact, only three levels out of eight can be used by user mode applications, the others are only available to the OS or the hypervisor. Modern Linux kernels running on POWER5 and POWER6 processors exploit software-controlled priorities in few cases such as reducing the priority of a process when it is not performing useful computation. Basically, the kernel uses thread priorities in three cases:

- The processor is spinning for a lock in kernel mode. In this case the priority of the spinning process is reduced.
- A CPU is waiting for operations to complete. For example, when the kernel requests a specific CPU to perform an operation by means of a *smp_call_function()* and it can not proceed until the operation completes. Under this condition, the priority of the thread is reduced.
- The kernel is running the idle process because there is no other process ready to run. In this case the kernel reduces the priority of the idle thread and eventually puts the core in Single Thread (ST) mode.

In all these cases the kernel reduces the priority of a hardware-thread and restores it to MEDIUM (4) as soon as there is some work to perform. Furthermore, since the kernel does not keep track of the actual priority, to ensure responsiveness it also resets the thread priority to MEDIUM every time it enters a kernel service routine (e.g., interrupt, exception handler or system call). This is a conservative choice induced by the fact that it is not clear how and when to prioritize a hardware-thread and what the effect of that prioritization is.

In order to explore the entire priority range, we develop a kernel patch that provides an interface to the user to set all the possible priorities available in kernel mode:

- We make priority 0 to 7 available to the user. As mentioned in Section 6.4.3, only three priorities (4, 3, 2) are directly available to the user. Without this kernel patch, any attempt to use other priorities result in a *nop* operation. Priority 0 and 7 (thread off and single thread mode, respectively) are available to the user through a hypervisor call.
- We avoid the use of software-controlled priorities inside the kernel, otherwise experiments would be effected by unpredictable priority changes.

- Finally, we provide an interface through the `/proc` pseudo file system which allows user applications to change their priority.

6.5.3 Running the experiments

To execute the experiments we use the FAME (FAirly MEasuring Multithreaded Architectures) methodology [163][162]. In [163] the authors state that the average accumulated IPC of a program is representative if it is similar to the IPC of that program when it reaches a steady state. The problem is that a benchmark has to run for a long time to reach this steady state. FAME determines how many times each benchmark in a multi-threaded workload has to be executed so that the difference between the obtained average IPC and the steady state IPC is below a particular threshold. This threshold is called MAIV (Maximum Allowable IPC Variation). The execution of the entire workload stops when all benchmarks have executed as many times as needed to accomplish a given MAIV value. For the experimental setup and micro-benchmarks used in this work, in order to accomplish a MAIV of 1%, each micro-benchmark must be repeated at least 10 times. In our experiments we run two micro-benchmarks, hence each experiment ends when both threads re-execute at least 10 times. Note that, at this point the fastest thread might already execute more than 10 times.

6.5.4 Micro-benchmarks description

In order to build a basic knowledge of the effect of software-controlled priorities, we used METbench (Minimum Execution Time Benchmark [34]), a micro-benchmark suite designed to stress specific processor characteristics.

Micro benchmark	**Class**
cpu_int, *cpu_int_add* *cpu_int_mul*, *lng_chain*	Integer
cpu_fp_asm	Floating Point
ldint_l1, *ldint_l2* *ldint_mem*	Memory

Table 6.4: Micro-benchmarks in METbench

We classify micro-benchmarks into three classes: Integer, Floating Point and Memory as shown in Table 6.4. In the Integer class there are *cpu_int*, which contains mixed integer instructions (one multiplication every two additions), *cpu_int_add*, which contains integer additions, *cpu_int_mul* which contains integer multiplications, and *lng_chain*, which is composed of mixed integer instructions with high inter-instruction dependency. The latter is designed to limit ILP exploitation for out-of-order processors (i.e. POWER5). In the Floating Point class, *cpu_fp_asm* is a benchmark that has a high percentage of mixed type floating point instructions. In the memory class, there are three micro-benchmarks: *ldint_l1*, *ldint_l2* and *ldint_mem*. Micro-benchmarks *ldint_l1* and *ldint_l2* are designed to always hit in the L1 and L2 cache, respectively, while *ldint_mem* is designed to always miss in cache.

All the micro-benchmarks share the same structure: they implement a `for` loop with enough iterations to run for at least one second. Hence, the micro-benchmarks differs mainly in the loop body, which shows a different instruction-mix according to the desired behavior.

We validated the behavior of each micro-benchmark through analyzing performance counters.

6.5.4.1 Integer micro-benchmarks

The four integer micro-benchmarks share a common loop structure. Listing 6.1 shows the main loop code for *cpu_int*. The code for *cpu_int_add* is the same except that in the loop there are only additions (for instance instead of *c=c*c*it* we used *c=c+c+it*). Analogously the loop for *cpu_int_mul* contains only multiplications. Note the macro LOOP_UNROL_100, used to repeat the same code 100 times, reducing control-flow instructions in the loop.

```

  for(it=0;it<micro_it; it++) {
    LOOP_UNROL_100(
    a=a+a+it; b=b+b+it; c=c*c*it;
    d=d+d+it; e=e+e+it; f=f*f*it;
    g=g+g+it; h=h+h+it; i=i*i*it;)
    }
```

Listing 6.1: *cpu_int* main loop

6.5.4.2 Floating point micro-benchmarks

In the Floating Point class, we implement the *cpu_fp_asm* micro-benchmark in POWER assembly in order to have a better control on its behavior, and hence maximize the use of the floating point unit.

6.5.4.3 Memory micro-benchmarks

The three memory micro-benchmarks share a common loop structure. In the loop, loads are executed using a *pointer chasing* technique. In this technique an array is initialized with pointers, so that each element contains the address of the next element to access. In order to execute several times, the last element of the array contains the address of the element at the beginning.

```

  for (i=micro_it; i>0; i=i-1) {
      LOOP_UNROL_100(p = *p;)
    }
```

Listing 6.2: *ldint_l1* main loop

Listing 6.2 shows the main loop code for *ldint_l1*. The code for *ldint_l2* and *ldint_mem* is exactly the same except that the array size varies in order to obtain the desired use of the cache hierarchy. Specifically, *ldint_l1* uses approximately 25% of the first level cache and makes all loads hit in L1. The micro-benchmark *ldint_l2* fills the first level cache, uses approximately 25% of the second level cache and makes all loads hit in L2. Finally, *ldint_mem* fills all the cache levels and makes all loads to access main memory.

6.6 Analysis of results

In this section we analyze the performance variation obtained with the software-controlled priority mechanism. First we analyze the performance of micro-benchmarks running in SMT mode with default priorities (priorities 4/4), then we analyze the malleability for threads running with higher and lower priorities. Subsequently, we show the effect of using the maximum priority difference in SMT (priorities 6 and 1). Finally, we show the malleability of benchmarks from SPEC CPU2006 suite.

6.6.1 Default Priorities

When running with default priorities (priorities 4/4) core resources are equally shared between threads. The default priority configuration is used to optimize throughput when knowledge about workload characteristics is not available. Threads running in SMT mode have lower performance compared to running in ST mode. Table 6.5 shows the average instructions per cycle (IPC) decrement of each micro-benchmark running in SMT mode with default priorities against all other micro-benchmarks, with respect to running in ST mode.

Table 6.5: Average IPC decrement of each micro-benchmark running in SMT mode against all the others, compared to ST mode.

micro-bench.	POWER5	POWER6
cpu_int	39.4%	28.0%
cpu_int_add	69.1%	23.8%
cpu_int_mul	18.7%	37.6%
lng_chain	23.8%	20.2%
cpu_fp_asm	27.1%	0.8%
ldint_l1	11.9%	9.3%
ldint_l2	14.2%	8.1%
ldint_mem	2.1%	0.01%

For example, the first row shows the average IPC decrement of *cpu_int* running in pairs {[*cpu_int*, *cpu_int*], [*cpu_int*, *cpu_fp_asm*], [*cpu_int*, *cpu_int_add*], ... [*cpu_int*, *ldint_mem*]} with respect to the IPC when running in isolation. We observe that:

- CPU intensive micro-benchmarks (*cpu_fp_asm*, *cpu_int*, *cpu_int_add* and *lng_chain*) show more IPC decrement when they run in POWER5 than when they run in POWER6.
- Micro-benchmarks with instruction dependencies or memory bounded microbenchmarks (ldint l1, ldint l2, ldint mem) show less significant IPC decrement and are quite similar in POWER5 and in POWER6.

Based on the results, our main conclusion are:

- CPU intensive micro-benchmarks that exploit out-of-order execution are more affected by SMT execution in POWER5 than in POWER6.
- micro-benchmarks with instruction dependencies and memory-bounded micro-benchmarks cannot fully exploit execution resources when in ST mode due to their intrinsic dependencies. Therefore, the two threads of the latter types can overlap and efficiently use the execution units in SMT mode.

6.6.2 Malleability

Let IPC_{ST} be the IPC that a given thread has when it runs in ST mode (single-thread mode [149]). In ST mode all core resources are allocated to the only running thread.

Let $IPC_{SMT}^{P/Q}$ be the IPC of the same thread when it runs in SMT mode with another thread, the first thread having priority P and the second thread having priority Q. For instance, $IPC_{SMT}^{4/4}$ is the IPC of that thread when it runs with another thread, both having priority 4 (default priority configuration).

We define *SMT thread malleability* (or simply *malleability*) as the ratio between the IPC in SMT mode with a given priority configuration and the IPC in SMT mode with default priority configuration:

$$\frac{IPC_{SMT}^{P/Q}}{IPC_{SMT}^{4/4}}$$

The highest IPC achievable by a thread is still IPC_{ST}, that is, for any priority configuration P/Q we have that: $IPC_{SMT}^{P/Q} \leq IPC_{ST}$. Hence, the malleability for a thread is upper-bounded by the IPC in ST mode normalized to the default priority configuration:

$$\frac{IPC_{SMT}^{P/Q}}{IPC_{SMT}^{4/4}} \leq \frac{IPC_{ST}}{IPC_{SMT}^{4/4}}$$

We consider that the maximum malleability is obtained using priorities 6/2, as we exclude priority 1 because it is designed for low-power executions. Figure 6.2 shows the correlation between the maximum malleability and the IPC in ST mode normalized to the IPC in SMT mode. Namely, x-axis reports $\frac{IPC_{SMT}^{6/2}}{IPC_{SMT}^{4/4}}$, y-axis reports $\frac{IPC_{ST}}{IPC_{SMT}^{4/4}}$, and each dot in the graph represents the actual pair of micro-benchmarks.

As Figure 6.2 shows, there is a clear positive correlation between these two variables (coefficient estimates: $b = 1.19$ and $R^2 = 0.97$). In fact, as explained before, the maximum performance that a task can obtain with priorities is upper-bounded by the ST performance.

6.6.3 Higher Priority

In this section, we analyze the malleability of a thread when it runs in SMT mode with higher priority than the other thread. We use priorities in the range 6-2 because priority 1 is used for low-power-mode, and it will be examined in detail in section 6.6.5.

Graphs in Figure 6.3 show a higher malleability on POWER5 compared to POWER6, when running CPU intensive micro-benchmarks. In POWER5 the thread speedup with higher priorities is up to 6 times, while in POWER6 it is less than 2 times. We can derive the following conclusions:

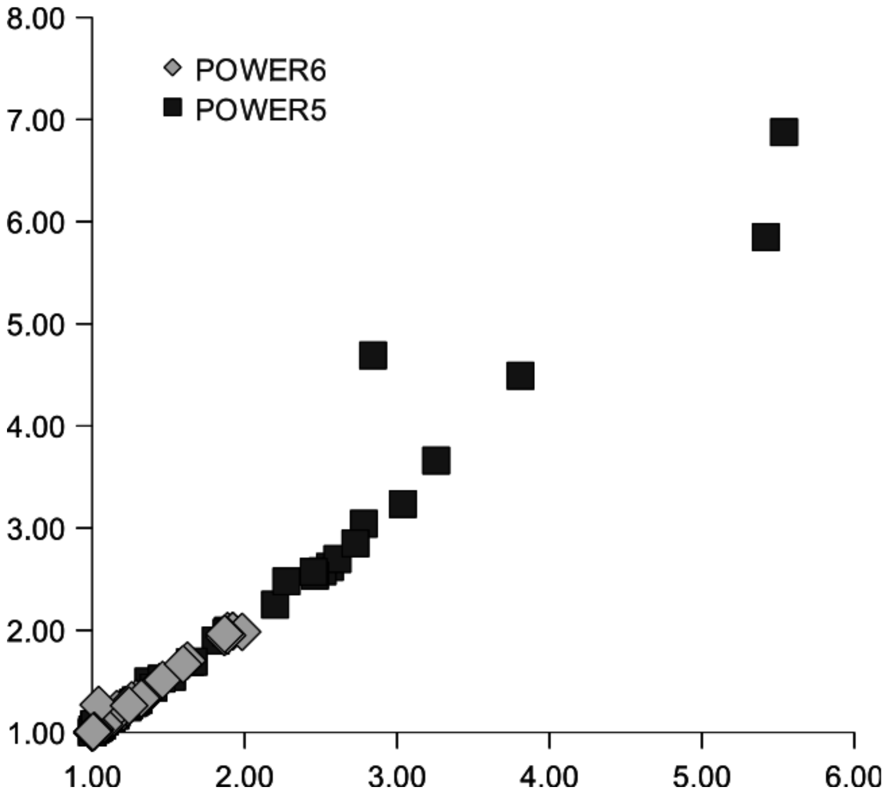


Figure 6.2: Correlation between the IPC in ST mode normalized to the IPC in SMT mode on y-axis ($\frac{IPC_{ST}}{IPC_{SMT}^{4/4}}$), and the malleability with priorities 6/2 on x-axis ($\frac{IPC_{SMT}^{6/2}}{IPC_{SMT}^{4/4}}$).

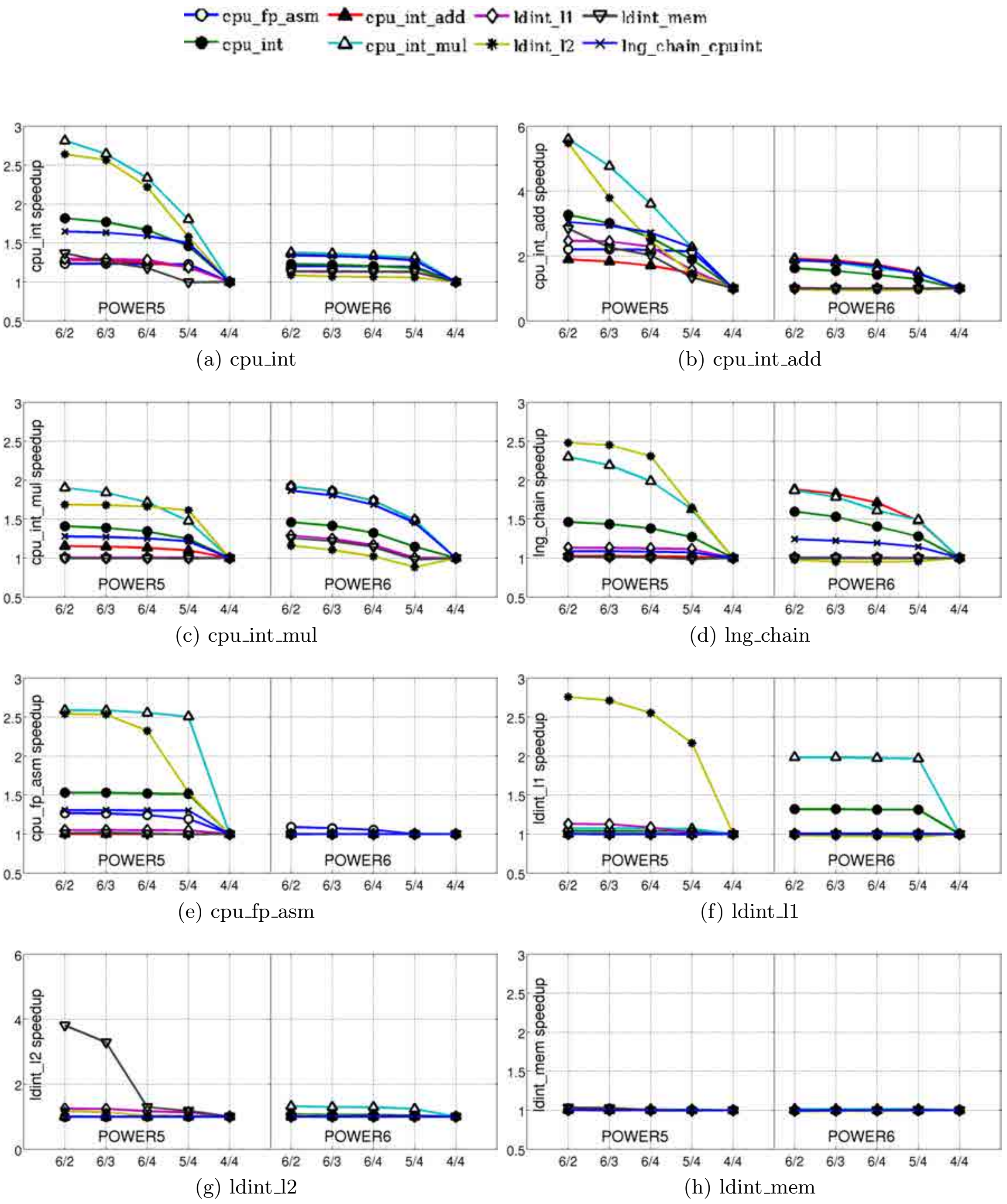
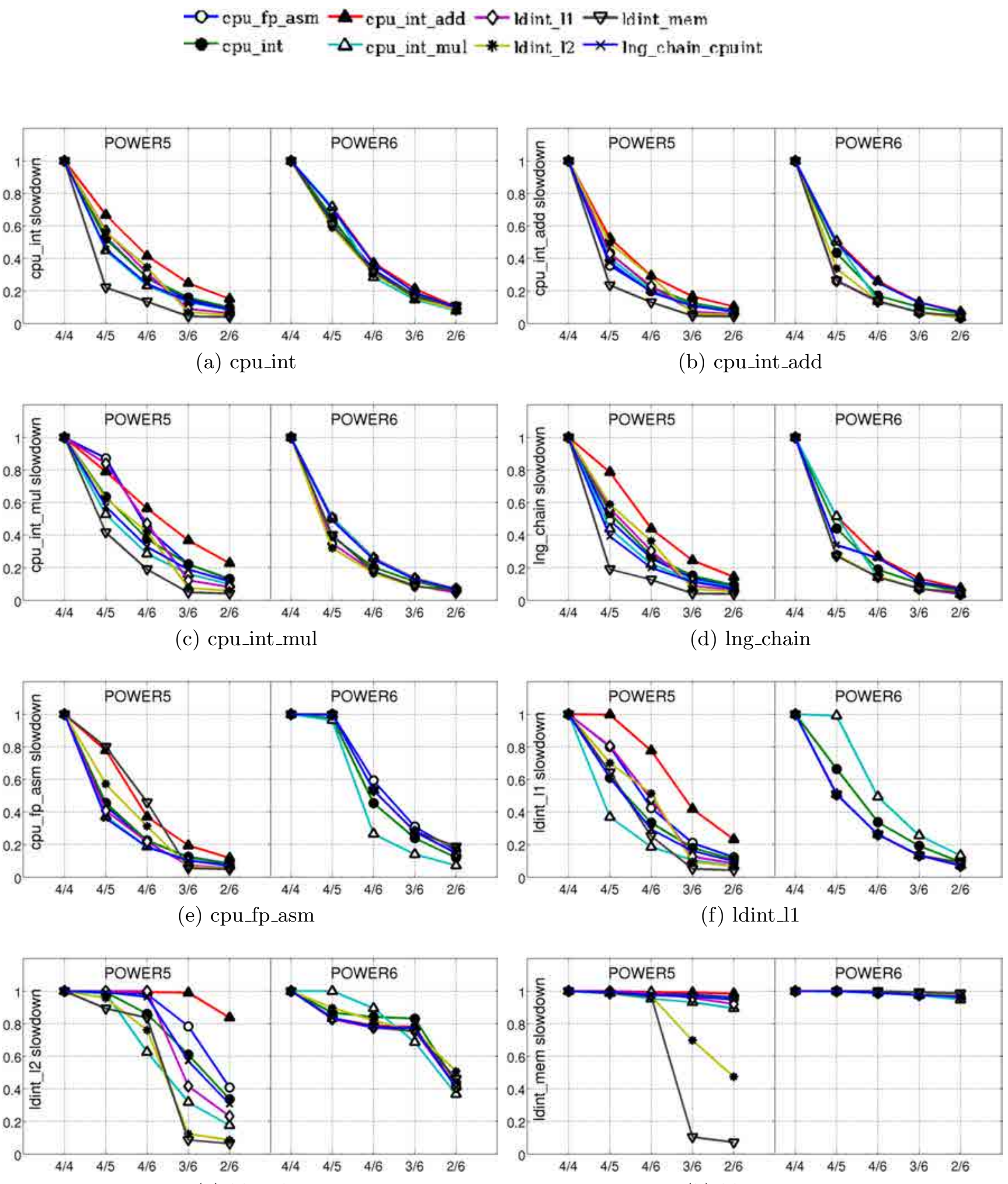


Figure 6.3: Malleability of the primary thread when its priority is higher than the priority of the secondary thread. Y-axis reports $\frac{IPC_{SMT}^{P/Q}}{IPC_{SMT}^{4/4}}$ and X-axis is the hardware priority for the primary and secondary threads (*primary-priority/secondary-priority*). Please note the different scale for *cpu_int_add* and *ldint_l2*



(a) cpu_int

(b) cpu_int_add

(c) cpu_int_mul

(d) lng_chain

(e) cpu_fp_asm

(f) ldint_l1

(g) ldint_l2

(h) ldint_mem

Figure 6.4: Malleability of the primary thread when its priority is lower than the priority of the secondary thread. Y-axis reports $\frac{IPC_{SMT}^{P/Q}}{IPC_{SMT}^{4/4}}$ and X-axis is the hardware priority for the primary and secondary threads (*primary-priority/secondary-priority*).

- The main reason for the lower impact of priorities on POWER6 is that the performance in SMT with priorities 4/4 are close to the upper-bound.
- In POWER5 we observe a high speedup in *cpu_int* (Figure 6.3a) and in *cpu_int_add* (Figure 6.3b) when their priorities are increased and they run with *cpu_int_mul*. The reason is that *cpu_int_mul* executes integer multiplications that take several cycles. Because the rate at which *cpu_int_mul* instructions complete is lower than the rate at which they are fetched into the processor, it clogs the issue queue. As a result, *cpu_int_mul* stalls the execution of CPU intensive micro-benchmarks like *cpu_int* or *cpu_int_add*. When we increase the priority of CPU intensive micro-benchmarks, their decode rate is higher, and hence they are less affected by *cpu_int_mul*.
- In POWER5 we can observe a high speedup when CPU intensive micro-benchmarks run with *ldint_l2* and when we increase CPU intensive micro-benchmarks priority (Figures 6.3a and 6.3b). The reason is that, with priorities 4/4, *ldint_l2* fills the load/store queue with high-latency loads, and prevents any other instruction from being dispatched. Consequently, using higher priorities for the CPU intensive micro-benchmarks when they run with *ldint_l2* results into a high speedup. However, the same behavior cannot be observed when CPU intensive micro-benchmarks run with *ldint_l1*, because load operations in *ldint_l1* have a lower latency and hence do not clog the load/store queue. This behavior is not observed when running with *ldint_mem* because of the automatic throttling mechanism [97] trigged by in-flight L2 misses. The same phenomenon happens when *ldint_l1* runs together with *ldint_l2* (Figure 6.3f). Finally, in POWER6 this phenomenon cannot be observed, because it is an in-order design. In fact, instructions in POWER6 can execute in the fixed point units even if the load/store queue is clogged. As a result, in POWER6 *ldint_l2* does not affect CPU intensive micro-benchmarks as it happens in POWER5.
- For the POWER6 we observe that the maximum speedup with priorities is obtained when executing two copies of micro-benchmarks using mainly a single functional unit (*cpu_int_add* in Figure 6.3b and *cpu_int_mul* in Figure 6.3c).

For *cpu_fp_asm*, *ldint_l1*, *ldint_l2*, *ldint_mem* in Figure 6.3 we observe that:

- In POWER6 for most of the micro-benchmarks the speedup is zero, because they reach the upper-bound performance (performance in ST mode) with priorities 4/4 (Figures 6.3e, 6.3g, and 6.3h).

- In POWER6 there is a speedup as we increase the priority of *ldint_l1* when it runs with *cpu_int_mul* (Figure 6.3f), because *ldint_l1* uses the fixed point unit to compute the effective address [105]. Since *cpu_int_mul* uses the fixed point unit with long latency operations, it competes with *ldint_l1* for this resource. As we increase the priority of *ldint_l1* it gets access to this resource more frequently, hence improving its performance. This is not observed in POWER5 (Figure 6.3f) because the effective address is computed through a dedicated adder inside the load/store unit.

- In POWER5, as we increase the priority of *ldint_l1* when it runs with *ldint_l2* we observe a high speedup (Figure 6.3f). This is because *ldint_l2* completely fills the L1 cache evicting the data of *ldint_l1*. By increasing the priority of *ldint_l1* we increase its cache access frequency, hence reducing the effect of *ldint_l2*. This speedup cannot be seen when *ldint_l1* runs with *ldint_mem*. The main reason is that *ldint_mem* has lower cache access frequency per cycle, as every load has to go to main memory. As a result, the cache lines belonging to *ldint_l1* are more frequently accessed and thus are not evicted by the least-recently-used (LRU) replacement policy.

- In POWER6 there is a small *ldint_l2* speedup because in most of the cases with priorities 4/4 we already reach the upper bound (ST mode).

- In POWER5 the speedup of *ldint_l2* running with *ldint_mem* is due to the fact that the *ldint_mem* fills completely the L2 cache, thus increasing the number of L2 misses of the former and hence reducing its performance.

- In POWER5 the speedup of *ldint_l2* running with itself is lower than when running with *ldint_mem*. Since *ldint_l2* uses only 25% of the L2 cache, two running *ldint_l2* can fit in the L2 cache. On the other hand, because *ldint_mem* completely fills the L2 cache, it considerably affects *ldint_l2*'s performance.

- In POWER5 and POWER6, *ldint_mem* (Figure 6.3h) is almost insensitive to a higher priority. This confirms the observation that micro-benchmarks with very low IPC cannot be improved using priorities, since with priorities 4/4 the upper bound (IPC in single-thread mode) is already reached.

6.6.4 Lower Priority

In this section, we present the malleability of a thread running with lower priority than the other thread. We consider the range of priorities 6-2, while priority 1, because of its special behavior (low-power mode), will be examined in section 6.6.5. Figure 6.4 shows that lower priorities significantly affects the performance of all micro-benchmarks.

Micro-benchmarks *cpu_int*, *cpu_int_add*, *cpu_int_mul*, *cpu_fp_asm*, *lng_chain* and *ldint_l1* in Figure 6.4 show that thread slowdowns are in the same order of magnitude in POWER5 and POWER6.

Micro-benchmarks *ldint_l2* and *ldint_mem* in Figure 6.4 show that in POWER6 lower priorities have a smaller impact than in POWER5. Note also the higher impact observed in POWER5 with priorities 3/6 and 2/6 (priority difference ≥ 3) when running with a memory bounded micro-benchmark. Furthermore, this behavior is not reported in POWER6.

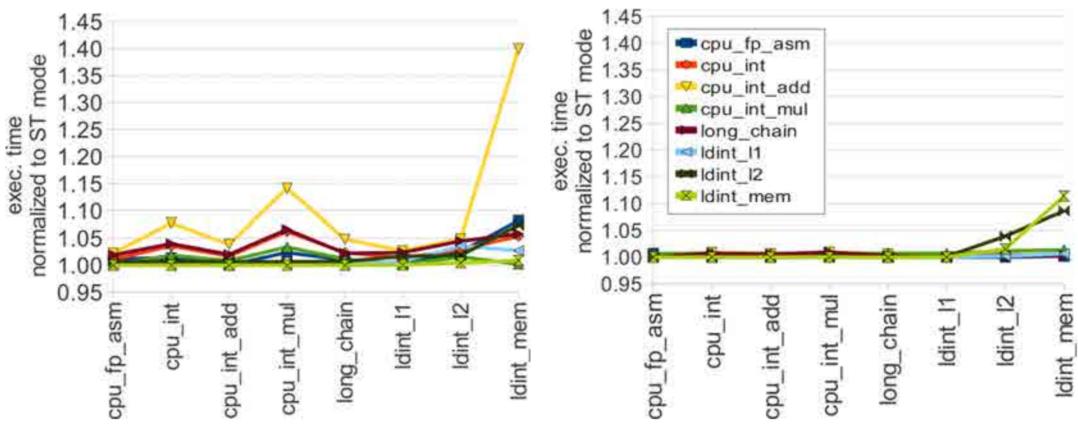
Based on the results we can conclude the following:

- Low-IPC micro-benchmarks are less affected by changing thread priority. For instance *ldint_l2* is less affected than *ldint_l1* and *ldint_mem* is less affected than *ldint_l2*.
- The use of lower priorities with memory-bound benchmarks leads to a smaller impact in POWER6 with respect to POWER5, this confirms the lower thread resource sharing in POWER6 microarchitecture compared to POWER5.
- In POWER5, a micro-benchmark running against *ldint_l2* or *ldint_mem* with priorities 3/6 and 2/6 (priority difference ≥ 3) shows a significative slowdown, while this cannot be observed in POWER6.

6.6.5 Maximum priority difference

The maximum priority difference in SMT is obtained when one thread has priority 6 and the other priority 1. The use of priorities 6/1 has an interesting effect: the thread with priority 6 shows a performance close to its single-thread mode. This result means that the priority mechanism can be used to provide an SMT configuration where we can run a background thread with minimum effect on the foreground thread.

Graphs in Figure 6.5 show the execution time (y-axis) of the primary thread when running with different secondary threads (x-axis) using priorities 6/1, for POWER5 and POWER6. Values are normalized to the primary thread ST execution time. In



(a) POWER5 (b) POWER6

Figure 6.5: Execution time of the primary thread running with priority 6 against a secondary thread with priority 1, normalized to the execution time in single-thread mode. X-axis is the actual secondary thread micro-benchmark.

POWER6 the performance impact on the primary thread is almost zero except when *ldint_l2* or *ldint_mem* runs with another memory-intensive micro-benchmark, mostly due to interactions at cache and memory levels. This shows that a thread can run in background without significantly affecting the primary thread.

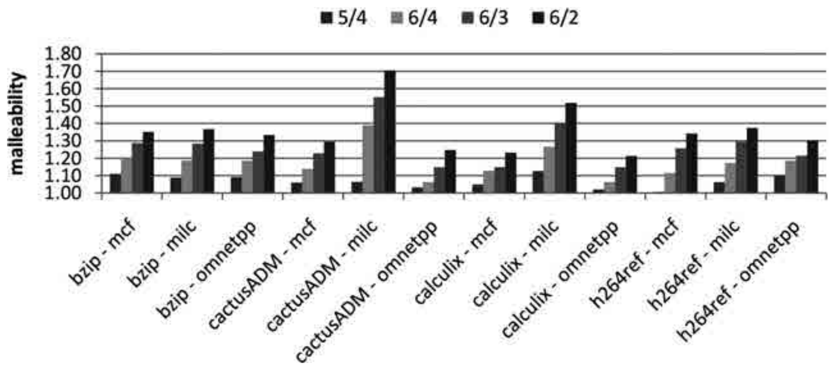
Table 6.6 shows the performance of the secondary thread when running with priority 1 as percentage of its single-thread performance. On POWER5, *ldint_l2* and *ldint_mem* achieve 19.09% and 86.57% of their single-thread performance while on POWER6, *ldint_l2* and *ldint_mem* achieve respectively 3.64% and 53.79% of their single-thread performance. For both machines, while CPU intensive micro-benchmarks report low performance with priority 1, *ldint_mem* maintains significant performance even when running with priority 1.

6.6.6 Malleability of SPEC CPU2006

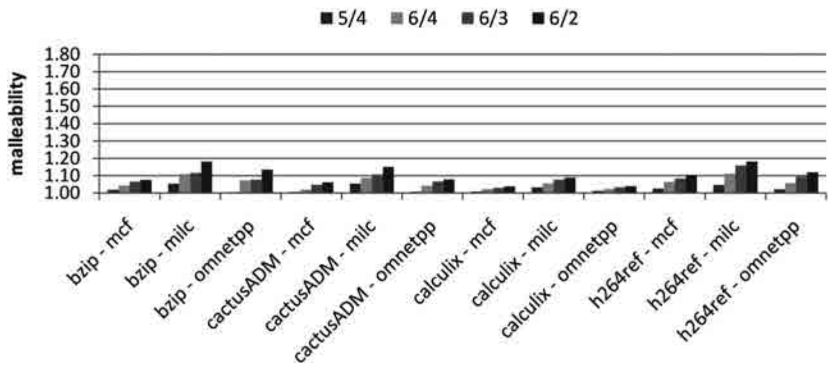
The two primary uses of software-controlled priorities are: providing imbalanced thread execution, as needed by the applications, and improving instruction throughput. In an imbalanced thread execution, software can control core resource allocation to improve a given target metric. For instance, enabling faster execution of higher priority jobs or implementing load balancing [34]. To achieve higher throughput, software can intentionally imbalance SMT resource sharing to improve the performance of the primary thread, without significantly reducing the performance of the secondary thread. For in-

Table 6.6: Performance of the background thread with respect to single-thread mode

micro-bench.	POWER5	POWER6
cpu_int	6.35%	1.63%
cpu_int_add	5.92%	0.89%
cpu_int_mul	2.59%	0.56%
lng_chain	9.71%	1.02%
cpu_fp_asm	7.09%	1.32%
ldint_l1	9.58%	1.07%
ldint_l2	19.09%	3.64%
ldint_mem	86.57%	53.79%



(a) POWER5



(b) POWER6

Figure 6.6: Malleability of selected SPEC CPU2006 using higher priorities for the primary thread.Y-axis reports $\frac{IPC_{SMT}^{P/Q}}{IPC_{SMT}^{4/4}}$ and x-axis is the actual pair of benchmarks running in SMT mode.

.

stance, when a CPU intensive thread is running together with a memory-bound thread, throughput can be improved by providing more resources to the CPU intensive thread.

In order to reduce hardware resource contention, high-IPC loads can be paired with low-IPC loads on the same core. As shown in previous sections, the effect of hardware priorities on memory-bound micro-benchmarks is smaller than the effect on CPU intensive micro-benchmarks. In this experiment, we run pairs in which the primary thread is high-IPC and the secondary thread is low-IPC. Based on the benchmark profile, we used bzip, cactusADM, calculix and h264ref as high-IPC benchmarks and mcf, milc and omnetpp as low-IPC benchmarks.

In this experiments we focus on the effects of higher priorities on the primary thread, assuming that the performance requirements of the secondary threads are subordinate to the performance requirements of the primary thread.

Figure 6.6 shows the speedup of the primary thread as we increase its priority with respect to the secondary thread. Using priorities 6/2 (primary thread priority is 6 and secondary thread priority is 2), the primary thread in POWER5 obtains a speedup up to 1.70 times the performance with default priorities, while in POWER6 up to 1.18 times the performance with default priorities.

Overall, hardware-thread priorities can be used when threads in a core present different hardware resource use. In particular, when the primary thread is CPU intensive and the secondary thread is memory-bound. In this situation, we increase the primary thread malleability without affecting the overall throughput.

6.7 Use cases

In this section we present two use cases of hardware thread priorities. Our objective is to show that, even if not for all kind of workloads, this feature can be effectively used to improve load balancing (use case A) and to implement transparent threads (use case B). The applications we use are taken from two different domains: a benchmark from the NAS Parallel Benchmarks [18] and six benchmarks from the CPU SPEC 2006 suite [86].

6.7.1 Use case A - Load Balancing

This use case shows how to use hardware thread priorities to reduce parallel applications' execution time.

Block Tri-diagonal (also called BT) is a benchmark from the NAS Parallel Benchmarks. BT is designed to solve discretized versions of the Navier-Stokes equation in three dimensions and uses a structured discretization mesh. BT Multi-Zone (BT-MZ) [94] is a version of the same benchmark that uses several meshes (also called zones) because often a single mesh is not enough to describe realistic complex domain. When BT-MZ runs both on POWER5 and POWER6 its MPI (Message Passing Interface) processes are imbalanced: during each iteration MPI processes have to wait for the last process to complete thus spending a significative fraction of time in waiting state, without performing any useful work.

To balance the application, tasks having high waiting time can be paired with tasks having low waiting time (bottlenecks), then scheduled on the same SMT core. Then, hardware thread priorities of tasks with low waiting time can be increased, to reduce the overall waiting time.

To balance BT-MZ, we run processes 1 and 4 on the first core and processes 2 and 3 on the second core. We found that the best combination of priorities is 4/5 for the first core and 4/6 for the second core. This configuration allows BT-MZ to be better balanced on both architectures.

Table 6.7: BT-MZ running on POWER5 with original and balanced configuration.

original configuration					
process	core	priority	running	waiting	others
1	1	4	24.61%	74.31%	1.08%
2	1	4	31.54%	67.23%	1.22%
3	2	4	58.54%	40.81%	0.64%
4	2	4	99.71%	0.13%	0.16%
execution. time: 66.80 sec					
balanced configuration					
process	core	priority	running	waiting	others
1	1	4	78.63%	20.91%	0.45%
2	2	4	65.88%	33.46%	0.65%
3	2	5	63.71%	35.72%	0.57%
4	1	6	99.78%	0.08%	0.15%
execution time: 59.97 sec					

Tables 6.7 and 6.8 show the breakdown of MPI states when BT-MZ runs with the original configuration and with the balanced configuration, on POWER5 and POWER6 respectively. The column *running* refers to the percentage of time the process is effectively running on the core, *waiting* refers to the percentage of time spent waiting for a synchronization and *others* refers to other MPI states with negligible contribution to

Table 6.8: BT-MZ running on POWER6 with original and balanced configuration.

original configuration					
process	core	priority	running	waiting	others
1	1	4	15.10%	83.84%	1.06%
2	1	4	25.25%	73.52%	1.23%
3	2	4	69.98%	29.44%	0.57%
4	2	4	99.56%	0.15%	0.29%
execution time: 40.05 sec					
balanced configuration					
process	core	priority	running	waiting	others
1	1	4	69.73%	29.51%	0.76%
2	2	4	63.24%	35.83%	0.94%
3	2	5	63.64%	35.69%	0.68%
4	1	6	99.57%	0.14%	0.29%
execution time: 34.32 sec					

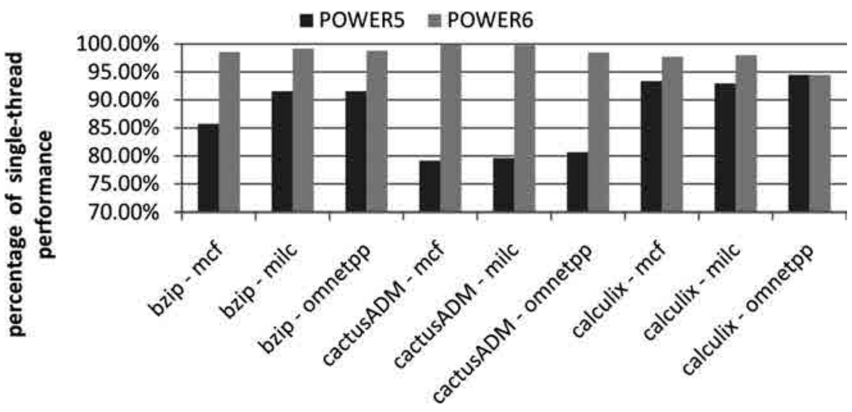
the total time. The percentage of time a process is in waiting state decreases when BT-MZ is executed with the balanced configuration. Consequently, the execution time is reduced by 11.4% on POWER5 and by 16% on POWER6.

6.7.2 Use case B - Transparent threads

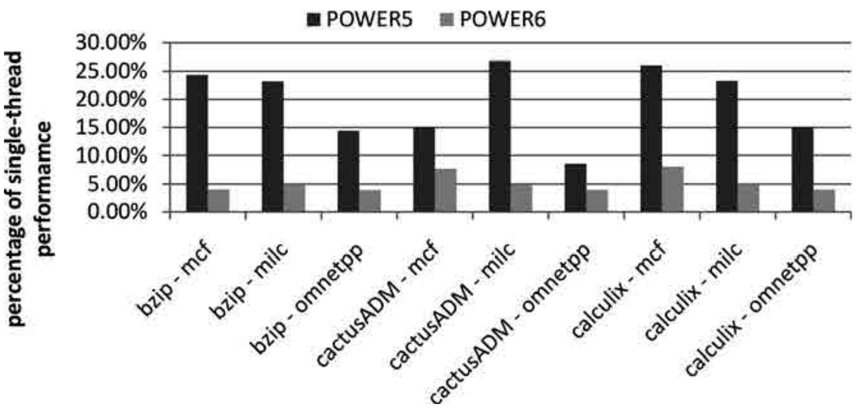
Dorai and Yeung [59] propose *transparent threads*: an SMT resource allocation policy that allows the background thread to use resources not required by the foreground thread. The objective is to obtain minimum performance degradation of the foreground thread compared to when it runs in single-thread mode.

In POWER5 and POWER6 this can be achieved using priority 6 for the foreground thread and 1 for the background thread. Potential uses are for instance in garbage collection, prefetching, virus scanning, file indexing, defragmentation or other low-priority kernel tasks.

The characterization with micro-benchmarks described in section 6.6.5 shows that transparent threading is more effective when the background thread is a memory bounded thread. To this extent we select six benchmarks from the SPEC CPU2006 benchmark suite: three CPU intensive to be used as foreground threads (bzip, cactusADM and calculix) and three memory bounded to be used as background threads (mcf, milc and omnetpp).



(a) foreground thread (priority 6)



(b) background thread (priority 1)

Figure 6.7: Transparent execution: percentage of the performance in single-thread mode for the foreground and the background threads. Y-axis reports $\frac{IPC_{SMT}^{P/Q}}{IPC_{ST}} \times 100$ and x-axis is the actual pair of benchmarks running in SMT mode with priorities 6 and 1.

.

Figure 6.7a reports the performance of the foreground thread using transparent thread execution with respect to its performance when running in isolation on POWER5 and POWER6. As shown in Figure 6.7a, the use of transparent threads is particularly effective on POWER6, with a performance degradation up to 5.5% for the selected benchmarks. On the other hand, due to the higher level of thread resource sharing, using transparent thread on POWER5 leads to a performance degradation of up to 20.86%. This result confirms the different effect of hardware thread priorities in POWER5 and POWER6 and lead to conclude that POWER6 architecture design is more adapt to exploit transparent execution.

Figure 6.7b reports the performance of the background thread using transparent thread execution with respect to its performance when running in isolation. As Figure 6.7b shows, the degradation of the background thread is considerable, especially on POWER6. This nonetheless should not be considered a drawback, given that the purpose of transparent execution is to run a thread in background that does not have performance requirements.

6.8 Conclusions

In this chapter, we characterize the software-controlled hardware-priority mechanism for IBM POWER5 and POWER6, based on the use of micro-benchmarks. We use a systematic approach in which we execute experiments with all the priorities combinations and with different running modes (ST and SMT). With this methodology we obtain several architectural insights that explain different behaviors of the thread prioritization mechanism on POWER5 and POWER6. The main conclusions are the following:

- The use of priorities generally leads to a smaller performance difference between ST and SMT modes in POWER6 than in POWER5, mostly due to the absence of the out-of-order execution on POWER6. Since in POWER6 the per-thread SMT malleability is smaller than in POWER5, increasing the priority of a thread generally leads to a smaller speedup than in POWER5.

- On both processors we have confirmed the correlation between high IPC and high sensitivity to priorities.

- In POWER5 with a priority difference greater or equal to 3 there is a significant malleability of the memory bounded threads. Therefore, performance tuning using priority differences greater or equal to 3 should be performed with a good understanding of the workload's memory behavior.

- We empirically measure the correlation of the malleability with the performance variation between SMT and single-thread execution.

- We show that hardware priorities can be used to improve load balancing for parallel applications: the execution of BT-MZ (NAS benchmarks) with a balanced configuration obtains an execution time reduction of 11.4% on POWER5 and of 16% on POWER6.

- We evaluate transparent execution, a mechanism that allows the foreground thread to run in SMT mode with performance close to single-thread mode. With applications from SPEC CPU2006 benchmark suites, the foreground thread reach up to 94% of its performance in single-thread mode running on POWER5, and up to 99% running on POWER6.

Overall, software controlled SMT priorities could be integrated into an adaptive runtime system to improve several metrics for a specialized class of applications.

Chapter 7

Exploiting cache locality and network-on-chip to optimize sorting on a Many-core architecture

This part of the thesis describes how to exploit user-level hardware features to be integrated into a specialized runtime system. In the previous chapter we examined the use of hardware thread priorities of IBM POWER5 and POWER 6 processors for regular and computation intensive applications. In this chapter we study a modern many-core processor and how its hardware characteristics could be effectively exploited to improve irregular and data-intensive applications. We choose the radix sort as a basic kernel of irregular and data intensive application. In the following sections we describe how the Tilera TILEPro network on chip and cache locality can be used to improve performance. As mentioned before, the hardware features studied in the following section can be effectively exploited by a specialized runtime system to speedup this class of applications.

7.1 Summary

The current trend in computer architectures indicates that Graphic Processing Units (GPUs) and manycore processors, integrating hundreds of simple cores, will be primary design points for power efficient High Performance Computing (HPC) systems. One of the challenges to efficiently implement programs such as radix sort on such architectures is how to extract parallelism from the algorithm. Radix sort passes are designed to be executed sequentially. During each pass, radix sort builds a global histogram using digits from all keys, thus synchronization on this structure is required for parallel execution. Moreover, as any sorting algorithm, it performs many memory operations with low data locality to move keys to their new positions.

Each architecture features specific characteristics that lead to different implementations and optimizations of the radix sort algorithm. On a shared-memory multicore processor, for instance, an efficient implementation should exploit data locality and use caches as a buffer to improve the irregular memory writes throughput [144]. GPUs and architectures featuring large SIMD units can leverage split operations [30] to locally reorder keys and improve overall performance. Manycore architectures featuring fast and dedicated Network-on-Chip (NoC) interconnects, can exploit more scalable core-to-core communications.

The Tilera architecture represents one of the first commercial examples of NoC-based manycore architectures. In particular, the TILEPro [9] family features from 36 to 64 tiles (simple RISC cores) on a chip. Tiles are interconnected through six NoCs, and feature a Dynamic Distributed Cache that improves coherent cache performance for large core counts. While the TILEPro processors lack floating point units and do not appear a good fit for classic scientific HPC applications, several vendors are proposing systems integrating them for many integer-based applications such as databases, networking, multimedia and cloud computing, claiming superior performance-per-watt than traditional multicore-based servers [11].

In this chapter we present an optimized implementation of radix sort for the Tilera TILEPro64 processor. We detail how we exploited the architectural features of the processor to increase the performance of the algorithm. We provide an in-depth analysis of the optimizations implemented for each phase of the algorithm, and discuss them in relation to the processor's sustained performance and bandwidth. We show the throughputs reached by the whole algorithm on several representative datasets (up to 132 million sorted keys per second on 240 million random 32-bit uniformly distributed keys). Finally, we show how our solution provides performance-per-watt comparable to current high performance architectures such as Intel x86 multicores and NVIDIA GPUs.

The work proceeds as follows. Section 7.2 describes the experimental environment. Section 7.3 describes the Tilera TILEPro64 processor architecture and provides its performance characterization. Section 7.4 explains the parallel radix sort and related work. Section 7.5 presents the implementation details of the radix sort algorithm on the Tilera architecture. Section 7.6 presents the algorithm optimizations for the TILEPro64 processor. Section 7.7 is about the experimental evaluation.

7.2 Experimental environment

Table 7.1 shows the experimental environment used in this chapter. The table includes the hardware system, the operating system, the communication library and the benchmarks used for the experiments.

CPU	Tilera TILEPro64 (1 socket - 64 tiles) at 866 MHz
Memory	64 Gigabyte of DDR RAM
Network	Ethernet (not used) - Network-on-chip
System size	single node
Operating System	Linux
Communication Library	Tilera Multicore Components API
Benchmarks	Microbenchmarks and Radix Sort

Table 7.1: Experimental environment - Tilera TILEPro64 system.

7.3 Tilera many-core processor

In this section we introduce the Tilera TILEPro64. Section 7.3.1 describes the processor architecture, and Section 7.3.2 provides an architectural characterization of its memory subsystem through the use of micro-benchmarks.

7.3.1 Processor architecture

The TILEPro64 is a manycore processor composed of 64 processing elements (called *Tiles*), interconnected through six independent NoCs. The topology of the NoCs is a two-dimensional mesh. The processor features four on-chip memory controllers and several I/O interfaces (PCIe, GbE, 10 GbE, Flex I/O). Each tile includes: a low-power 3-way VLIW 32-bit processor engine running at 866 MHz; a cache engine containing a direct-mapped 16KB instruction L1 cache, a 2-way associative 8KB data L1 cache and a 4-way associative 64 KB L2 cache; a NoC switch engine for 6 functional independent networks. Each tile also contains a 2D DMA engine that orchestrates memory data streaming between tiles and external memory, and among tiles. The six NoCs include the I/O Dynamic Network (IDN), which directly connects all the tiles with the I/O interfaces, the Static Network (STN), which can be programmed to perform static routing with predefined end-points, the User Dynamic Network (UDN), which implements a pipelined distributed dimensional (X-Y) routing and can be used to perform tile-to-tile message passing at the user level, and three memory networks that communicate with the memory controllers and manage cache coherency.

The Tilera Multicore Components API (TMC) provides primitives that allow fine-grained control on many architectural features of the processor. The cache subsystem of the TILEPro architecture is very flexible and fully configurable. It is named Dynamic Distributed Cache (DDC). The L1 caches are private for each tile. Instead, the L2 cache is managed as a hybrid cache. A cache line is said to be “locally homed” in a tile if when there is a miss in the local L2 cache, the request is directly sent to the DDR memory. On the other hand, a cache line is said to be “remotely homed” if on a L2 miss, a request is sent to the “home tile”. In the latter case the home tile’s cache acts like a L3 cache (with an increased access latency) for tiles that miss on their own L2. Lines are then copied in the remote tile’s cache. The L2 caches are coherent, but coherency can be disabled through a software switch. The TILEPro also supports a hash-for-home policy, where pages are homed on different tiles’ L2s at cache line granularity (64 bytes). The developer can select which tiles participate in this strategy. Finally, the programmer can allocate memory pages either on memory banks managed by a specific memory controller or in striping mode, where each page is striped across the various memory controllers in chunks of 8 KB.

7.3.2 Architectural characterization

In this section we characterize the memory bandwidth of the TILEPro64 through a set of micro-benchmarks. This allows understanding how far from the maximum bandwidth the various phases of the radix sort are and how effective the optimizations are. We look at four parallel memory access patterns: sequential read and write (benchmarks *read-seq* and *write-seq*), and random read and write (benchmarks *read-rand* and *write-rand*). The benchmark *read-seq-pref* exploits prefetching and loop-unrolling to increase the performance of sequential reads. We also evaluate the effects of employing the hash-for-home (*Hash*) and Local homing (*Local*) policies. The benchmarks use from 1 to 62 threads. Each thread is pinned to a specific tile. Two tiles handle I/O operations and are not available to applications. Each thread allocates a chunk of memory and reads it or writes 32 bit integers into it.

Figure 7.1 shows the bandwidth for *read-seq*, *read-rand*, and *read-seq-pref* with the two different memory homing policies. As the figure reports, the bandwidth with the local homing policy is significantly higher than with the hash-for-home (except

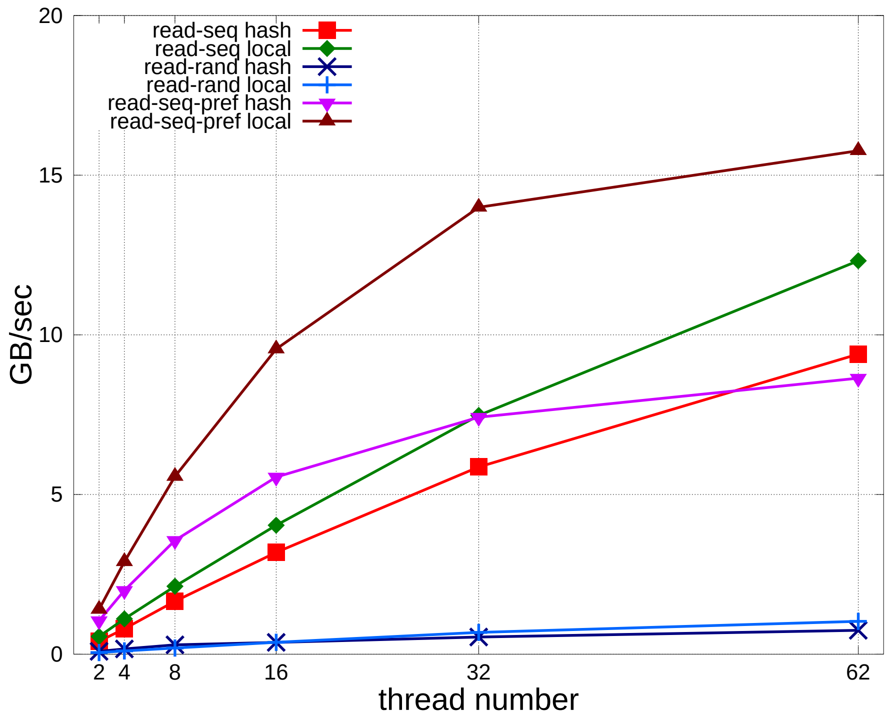


Figure 7.1: Read bandwidth.

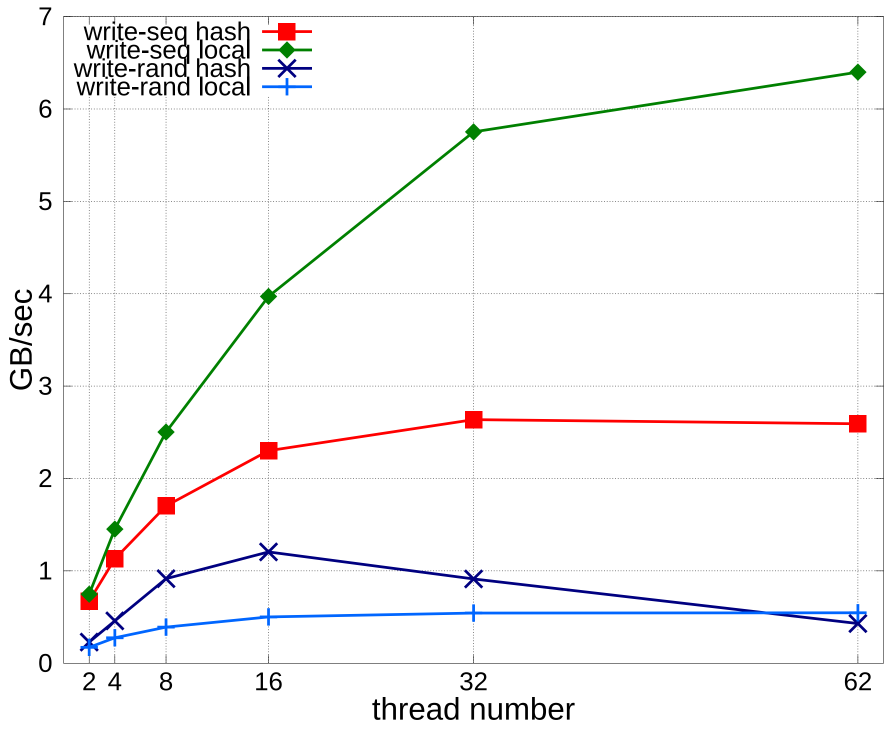


Figure 7.2: Write bandwidth.

for *read-rand*), due to the improved data locality on the allocating tile. Also, results indicate that loop-unrolling and prefetching are essential, leading to a higher bandwidth utilization.

Figure 7.3 shows the bandwidth for *write-seq* and *write-rand* with the two different memory homing policies. For sequential writes, the behavior is easily understandable. Since with local homing each tile owns the memory pages it is writing to, other remote caches are not queried. With hash-for-home, instead, each line is homed on a different cache, thus, whenever a new line is written, a different tile's L2 cache is accessed. For random writes the behavior is more complex. For low numbers of tiles the the hash-for-home policy is better. The reasons are two. First, since the memory accesses are completely random, the hashing allows for a better balance of the cache and memory NoCs access patterns. Second, since the pages are hashed all over the tiles of the system, even if only some of the tiles are effectively emitting memory operations, a virtually larger cache is used and the cache thrashing is reduced with respect to the local homing policy. However, when the number of tiles increases, total cache sizes become the same for both policies, and the higher irregularity of the access patterns in the NoCs with the hash-for-home policy reduces the bandwidth.

To further investigate the effects of random writes, we implemented a set of micro-benchmarks that write sequential chunks of 16 (*block-write-16*), 32 (*block-write-32*), 64 (*block-write-64*) and 128 (*block-write-128*) bytes in random memory positions. Figure 7.3 reports the bandwidth for each of the mentioned micro-benchmarks. As expected, *block-write-16* has the lowest bandwidth, due to the lowest data locality of the writes. Performing writes with larger sizes (*block-write-32*, *block-write-64* and *block-write-128*) considerably improves bandwidth utilization. Also, we can observe that local homing is the most performing memory allocation policy for most of the cases.

7.4 Background

This section provides preliminaries to understand the material presented in the rest of the work. Section 7.4.1 explains the reference serial radix sort implementations used in this work. Section 7.4.2 discusses the related work.

7.4.1 Radix sort

Radix sort sorts data by grouping keys by the individual digits which share the same significant position (radix). At each step, the grouping of the digits is usually done through Counting or Bucket Sort. Radix Sort can operate starting from the least

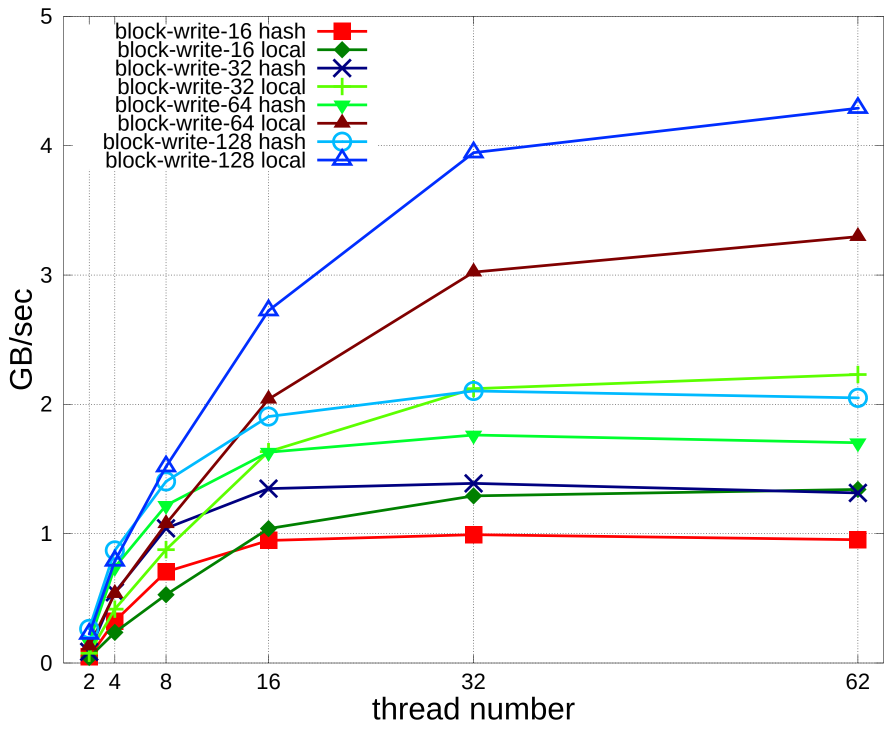


Figure 7.3: Write bandwidth for larger chunks of data.

significant digit and moving towards the most significant digit (LSD Radix Sort) or the other way around (MSD Radix Sort).

In the first step, Radix Sort divides the bits of each key in k groups of size l bits called radixes, e.g., 4 radixes of 8 bits form a 32-bit integer key. Then, for each radix, the algorithm performs a pass, ordering all the keys according to the values of that radix with the following steps: (1) compute the histogram of occurrences of each l-bit digit for the current radix; (2) compute the offset of each keys to order them by the current radix; (3) reorder the keys based on the offset.

After k passes, the algorithm produces a total ordering of the keys. While comparison-based sorting algorithms are lower-bounded by computational complexity of $O(n \times log(n))$, Radix Sort computational complexity is $O(nk)$, where k is the number of radixes (for fixed-lenght keys). The number of radixes can vary from 1 to the number of bits used to represent the keys to be sorted. There is a trade-off between the number of passes and the size of the histogram (2^l).

7.4.2 Related work

Sorting algorithms have attracted a large deal of research, due to their apparently simple statement but complexity in finding efficient solutions. Platforms based on multicore processors, manycore processors and GPUs have become interesting targets for these algorithms. Lately, many works discussing parallel implementations of radix sort on these architectures have appeared. Sohn and Kodama [152] present a balanced parallel radix sort that first obtains the bin counts of all the processors, and then computes for each processor the number of keys, their source bin and their source processor. Partitioned parallel radix sort [108] further improves upon the previous work by reducing the multiple rounds of data redistribution to one. In [84] a MSD radix sort is used to evaluate scatter and gather primitives for GPUs.

Satish et al. [143] introduce a merge sort and a radix sort for GPUs. The implementations exploit efficient data-parallel primitives such as parallel scan and the high-speed on-chip shared memory provided by NVIDIA's GPU architectures. In particular, for the radix sort implementation, the initial input is divided in tiles, that are assigned to different threads blocks and sorted in the on-chip memory using b iterations of 1-bit split operations [30]. The split operations are Single Instruction Multiple Data (SIMD) friendly, as they can be executed with a SIMD prefix sum and a SIMD subtraction. The radix sort implementation reaches a saturated performance of around 200 Mkeys/s for floating point keys on a GeForce GTX 280.

In [144] the previous SIMD-friendly approach for GPUs is compared to a buffer based scheme for CPUs that collects elements belonging to the same radix into buffers in local storage. Buffers are written from local storage to global memory only when enough elements have been accumulated. The buffer based scheme implementation reaches up to 240 Mkeys/seconds on a Core i7 (Nehalem architecture) processor.

In [145] the Intel Many Integrated Cores (MIC) architecture is also introduced. The SIMD-friendly approach for GPUs is adapted to the Knights Ferry implementation of MIC . The authors show saturated performance respectively at about 325 Mkeys/s for the GeForce GTX 280, 240 Mkey/s for the Core i7 and 560 Mkeys/s on Knights Ferry.

Merrill and Grimshaw [117] present a family of very efficient parallel algorithms for radix sorting on GPUs. The proposed algorithms exploit an efficient parallel prefix scan "runtime" that includes kernel fusion, multi-scan for performing multiple related, concurrent prefix scans and a flexible algorithm serialization that removes unnecessary synchronization and communication within the various phases. The algorithms proposed are included in the NVIDIA Thrust Library and have a sustained performance over 1000 Mkeys/s for the GeForce GTX 480 (Fermi architecture).

Wassenberg and Sanders [165] introduce a radix sort algorithm that builds upon a microarchitecture-aware counting sort. It exploits virtual memory, by generating as many containers as there are digits and inserting each value in the appropriate container. It exploits write combining by accumulating the values in the appropriate positions in buffers and then writing them directly to memory with non temporal stores that bypass all the data caches. On a dual Xeon W5580 (Nehalem architecture, 4 cores, 3.2 GHz) this algorithm reaches 657 Mkeys/s. However, the extensive use of virtual memory limits the maximum size of datasets.

7.5 Parallel Radix Sort

To parallelize radix sort, we divide the unsorted array of N keys into M blocks, where M is equal to the number of threads. Each block $block_t$ is assigned to a thread with rank t (in the following, we use t to denote the rank of both threads and blocks). A second array of size N stores the sorted keys at each pass. This array is also divided into M blocks, where each $block\_sorted_t$ is assigned to the thread t. Algorithm 1 shows the main loop of the parallel radix sort.

Algorithm 1 Parallel radix sort main loop.

```
for $pass = 0 \dots P$ do
    $hist_t \leftarrow$ DO_LOCAL_HISTOGRAM($block_t, pass$)
    $offset_t \leftarrow$ COMPUTE_OFFSET($hist_t$)
    DISTRIBUTE_KEYS($offset_t, block_t, pass$)
    $block_t \leftarrow block\_sorted_t$
end for
```

P is the number of passes, which corresponds to the number of radixes. At each pass, thread t performs the following phases:

1. Computes the histogram of occurrences for $block_t$ (DO_LOCAL_HISTOGRAM()).

2. Computes the new position for the keys in $block_t$ with a global parallel operation (COMPUTE_OFFSET()).

3. Writes the keys from $block_t$ into $block\_sorted_{t'}$ array, where $t' \in [0, M-1]$ can be a block owned by any thread (DISTRIBUTE_KEYS()).

4. Then, replace $block_t$ with $block\_sorted_t$ to prepare for the next pass.

Algorithm 2 Local histogram computation.

```
function DO_LOCAL_HISTOGRAM($block_t, pass$)
    for $i = 0 \dots B$ do
        $k \leftarrow block_t[i]$
        $d \leftarrow$ GET_DIGIT($k$,$pass$)
        $hist_t[d] \leftarrow hist_t[d] + 1$
    end for
end function
```

7.5.1 Local histogram

In this phase each thread reads its block of keys and, for each key in the block, extracts the proper radix and increments the histogram of occurrences. Algorithm 2 describes the different steps of this phase. $B = N/M$ is the number of keys in a block. The function GET_DIGIT() performs a mask operation to extract the digit from the number, according to the current pass. At the end of this phase, $hist_t$ contains the occurrences of all digits in $block_t$.

7.5.2 Offset computation

The purpose of this phase is to compute the offset, in the sorted array, for each key of the block. The offset is used in the next phase by thread t to distribute the keys from $block_t$ into $block\_sorted_{t'}$ (where $t' \in [0, M-1]$). We define $offset_t[d]$ as a local array that stores the new position of a key that has digit d and originally resides in $block_t$. The offset is expressed with respect to the entire array of keys, i.e. with respect to the first position of $block\_sorted_0$. Moreover, keys with the same digit for the current pass are stored in sequential positions, giving precedence to threads with lower index. The offset for keys in $block_t$, with digit d, is computed by thread t as:

$$offset_t[d] = \sum_{t'<t} hist_{t'}[d] + cumulative\_hist[d-1] \tag{7.1}$$

where $hist_{t'}$ is the local histogram of blocks with lower rank than the current one ($t' < t$) and $cumulative\_hist$ is the cumulative global histogram. The cumulative histogram, common to all the threads, accounts for all the digits smaller than d, while the precedence rule for threads with index lower than t guarantees that the computed offset is correct. To obtain the two operands of the sum in formula 7.1 we perform a parallel prefix sum operation starting from local histograms. The schema in Figure 7.4 describes a parallel prefix sum involving eight threads. Each rectangle represents a thread and $h0$, $h1$, ..., $h7$, refer to the local histograms. At each iteration, a partial sum of the local histograms is forwarded to the threads with higher rank. At the end of the third iteration (iter $= 3$), each thread t has the sum of the local histograms from the threads $t' < t$ and the thread with the highest rank has the global histogram. Theoretically, this approach is not the most efficient way of performing a parallel prefix sum. We explain this design choice in section 7.6.2.

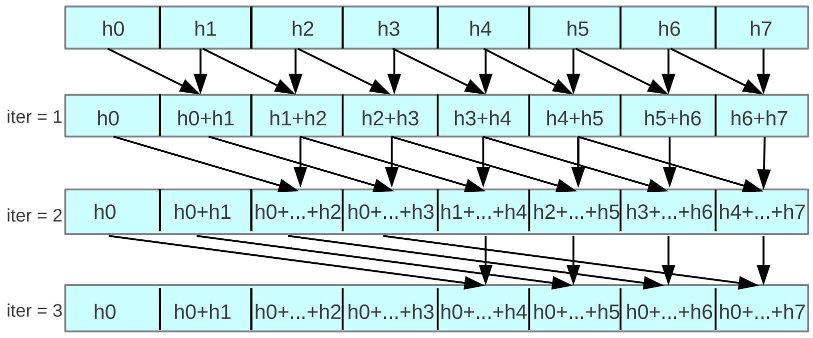


Figure 7.4: Parallel prefix-sum with 8 threads.

7.5.3 Keys distribution

In this phase, each thread writes keys from block $block_t$ to $block\_sorted_{t'}$ (where $t' \in [0, M - 1]$). Algorithm 3 describes the steps performed by a thread for each key. The thread reads the key from the local block and extracts the digit d according to the current pass. Then, it writes the key in a position obtained by $offset_t[d]$ into $block\_sorted_{t'}$. Finally it increments $offset[d]$ so that the next key with digit d is written into the following position. Since the new position of the key o is expressed as the offset from the beginning of the first block ($block_0$), the destination block (t') and the relative offset ($o \ mod \ B$) are also computed. The division to obtain the block number ($t' \leftarrow o \ / \ B$) is rounded towards zero. At the end of this phase, the keys have been partially reordered according to the current radix.

7.6 Optimizations

This section describes the optimizations to the radix sort implementation previously presented. Although each optimization has been evaluated with the all datasets presented in Section 7.7.1, in this section we show their effects with a uniform distribution of 240 million random 32-bit integers.

Algorithm 3 Keys distribution.

```
function DISTRIBUTE_KEYS($block_t, offset_t, pass$)
    for $i = 0 \dots B$ do
        $k \leftarrow block_t[i]$
        $d \leftarrow$ GET_DIGIT($k$,$pass$)
        $o \leftarrow offset_t[d]$
        $t' \leftarrow o \mathbin{/} B$
        $block\_sorted_{t'}[o \bmod B] \leftarrow k$
        $offset_t[d] \leftarrow offset_t[d] + 1$
    end for
end function
```

7.6.1 Local Histogram

As mentioned in Section 7.4, the histogram size H depends on the size of the radix l: $H = 2^l$. The radix size significantly affects performance, because it determines the size of several data structures and the number of radix sort passes. We found that the best trade-off for our implementation is using five passes with the following radixes: 7, 7, 6, 6, 6. This determines local histograms of 512 bytes (128 elements of 32 bits) for the first two passes and 256 bytes the other three passes. Because of the small sizes, the local histograms likely reside in the caches. Since the update of a local histogram corresponds to reading a chunk of sequential data and incrementing values in the cache, the upper-bound for the performance of this phase is the memory bandwidth utilization for sequential reads.

We implemented two optimizations to improve the performance of this phase: local block caching and data prefetching. The first optimization leverages the flexibility of Tilera's caches by allocating the blocks of the input array and the histograms with the *local homing* policy. This allows exploiting data locality at the tile level (a thread is pinned on a tile), improving access latency and reducing memory traffic. The second optimization leverages prefetching and loop-unrolling to increase bandwidth utilization for contiguous reads from memory, as described in Section 7.3.2. We use the TMC library function *tmc_mem_prefetch()* to prefetch chunks of data into the cache before they are actually read. To fully exploit prefetching, we unroll the read loop in multiples of the cache line size (64 bytes or 16 elements).

Figure 7.5 shows the bandwidth utilization for the unoptimized (*hist*) and the optimized (*hist-pref*) implementations of this phase. As the figure shows, the optimizations for this phase lead to a bandwidth improvement of 1.68 times. As expected, the bandwidth of *hist-pref* is close to the bandwidth of the micro-benchmark *read-seq-pref* of Section 7.3.2.

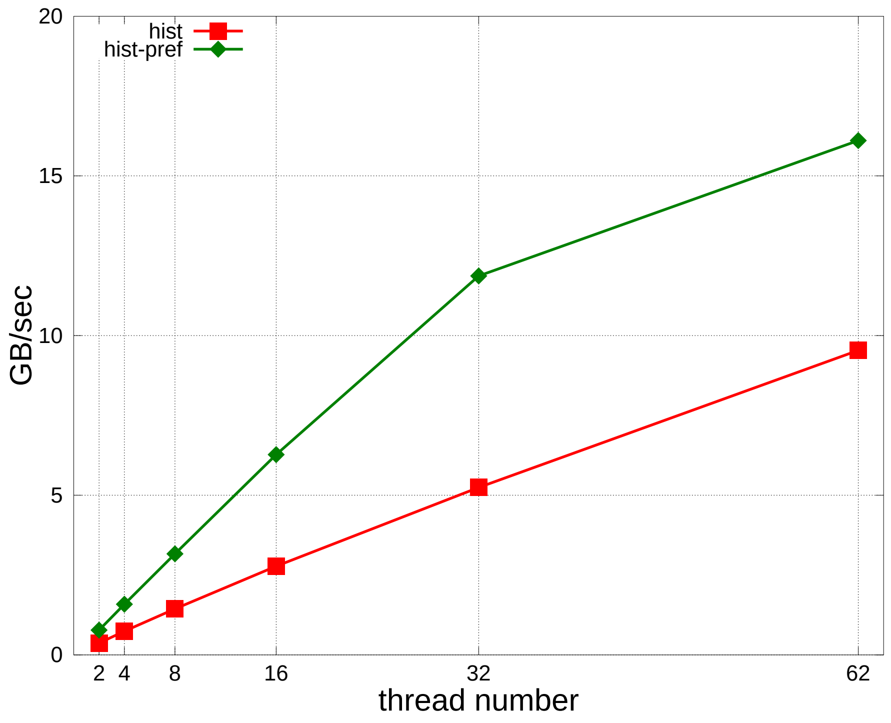


Figure 7.5: Histogram computation bandwidth.

7.6.2 Offset computation

The offset computation phase performs a prefix sum using the local histograms, and does not involve the original dataset (keys). Since the local histograms fit in caches, this is the only non-memory bounded phase. Our first implementation of this phase was a shared-memory parallel prefix sum algorithm based on [30] (*prefix-sum-shared*). To further improve this phase, we implemented a message passing version of the parallel prefix sum as proposed by [88] (*prefix-sum-udn*), using the UDN (the NoC for user-level inter-tile communication). Figure 7.6 reports the execution times of the two implementations. Since the prefix sum is performed using local histograms with a constant size per thread (512 bytes), the execution time increases when increasing the number of threads. As the figure shows, with 62 threads the UDN-based prefix sum is 2.57 times faster than the shared-memory version, even if the former has a higher computational complexity.

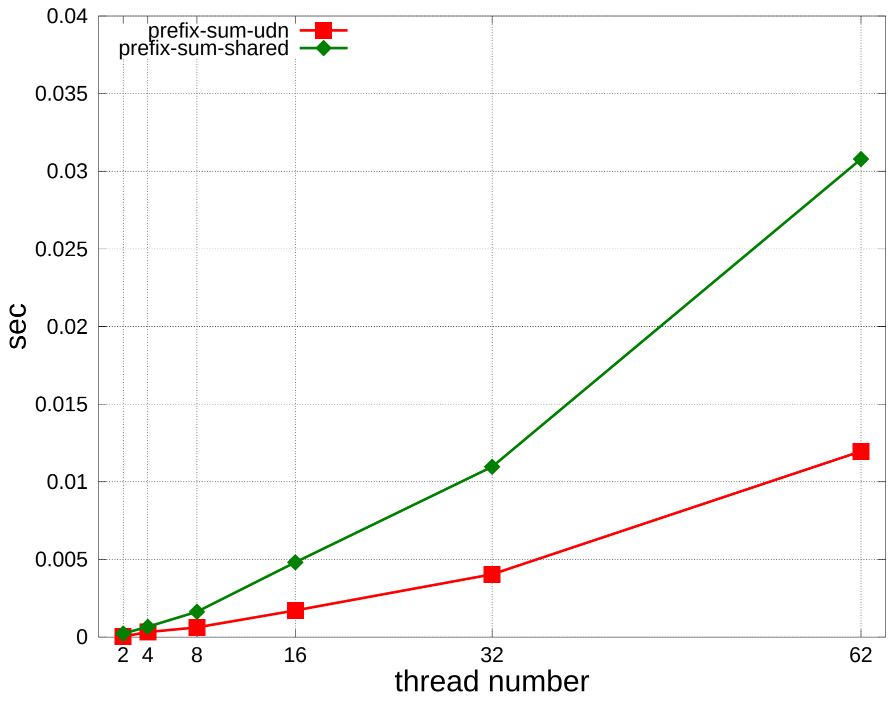


Figure 7.6: Prefix sum.

7.6.3 Keys Distribution

Even if the computational complexity of this phase is ($O(n)$), the presence of high-latency non-contiguous memory writes makes it the bottleneck of the algorithm. In fact, write locations for each key, that threads obtain from $offset_t[d]$, are independent and most likely fall into un-cached portions of $block_sorted$. The probability of having two contiguous writes is inversely proportional to the size of the dataset, hence it is very low for large datasets. Since this phase is the bottleneck of the radix sort, its optimizations have the highest impact on the overall algorithm performance. The memory writes of this phase are analogous to the *write-rand* operation described in section 7.3.2.

The first optimization is related to the starting address of the blocks. To exploit local homing, each thread must allocate its own block. Every time a thread writes to a block, not necessarily its own, it must retrieve its starting address (e.g., from a global structure). However, in our implementation blocks are allocated contiguously in memory, through the function *tmc_alloc_map_at()*. This function allows specifying the virtual address of each block. Because blocks are adjacent in memory, and all have the same size, every memory location can be obtained by computing an offset from the

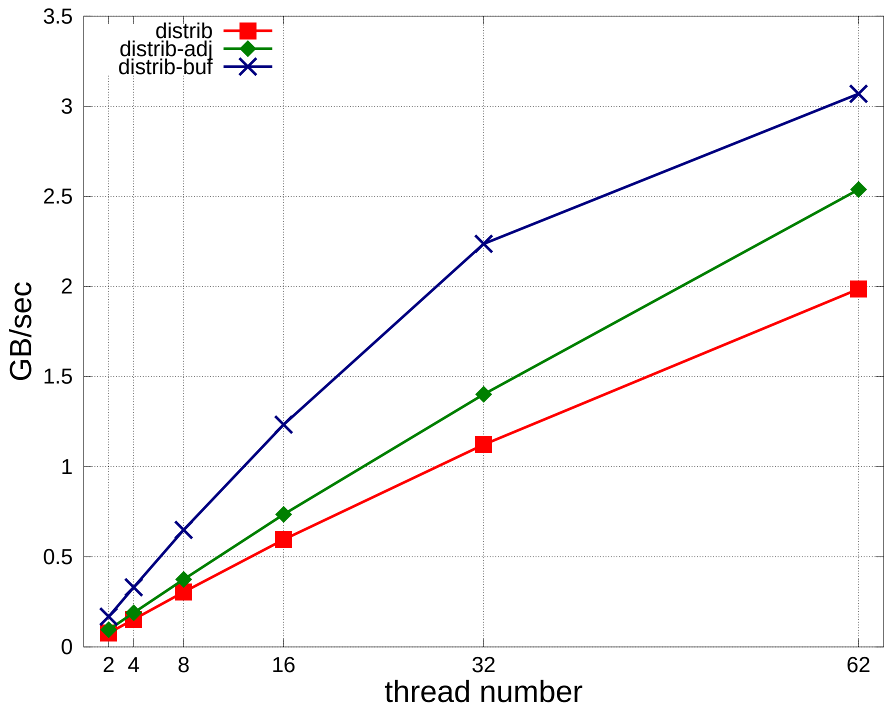


Figure 7.7: Key distribution bandwidth.

starting address of the first block. For example, position 8 of block 3 can be accessed by adding $8 + (3 \times B)$, where B is the number of elements in a block, to the starting address of block 0.

Furthermore, as shown in section 7.3.2, random write bandwidth can be improved by writing chunks of data larger than 4 bytes. Hence, the second optimization is to implement a write buffer to leverage larger memory writes. This optimization exploits the observation that keys with the same digit are going to be written into consecutive positions. Because each block is allocated using the local homing strategy, keys are most probably going to be written into blocks that are remotely cached. Thus, it is convenient to first accumulate keys with the same digit into the local cache (possibly L1) and then write them all together to memory. The write buffer is allocated into the local stack, that uses the local homing strategy by default.

Figure 7.7 reports the bandwidth utilization for the keys distribution phase: *distrib* is the basic key distribution phase without optimizations, *distrib-adj* is the key distribution phase with adjacent blocks in memory and *distrib-buf* uses adjacent blocks with a write buffer of 8 elements. Allocating adjacent blocks improves the bandwidth up to 1.28 times with 62 threads. The write buffer of 8 elements, together with the adjacent block allocation, lead to a performance improvement of 1.55 times. As expected, the bandwidth of *distrib-buf* is similar to the bandwidth of the micro-benchmark *block-write-64 local* shown in Figure 7.3 that performs similar memory operations (a write of 8 elements corresponds to 64 bytes). We found that using a write buffer larger than 8 elements causes a performance degradation. If we compute the buffer size (given a digit size of 7 bits, the size of a buffer for 8 elements is $128 * 4 * 8 = 4096$ byte) it is easy to observe that a larger buffer would fill the L1 data cache (8192 KB), causing conflicts with other structures used in the algorithm.

7.7 Experimental evaluation

This section discusses the experimental results of our radix sort implementation for the Tilera TILEPro64 processor. Section 7.7.1 presents the experimental setup. Section 7.7.2 presents the performance improvement obtained with the optimizations. Section 7.7.3 analyzes the scalability of our implementation. Finally, Section 7.7.4 compares the implementation for TILEPro64 to implementations for other architectures.

7.7.1 Experimental setup

The TILEPro64 is hosted on a TILEmpower platform, which offers 16GB of DDR2 memory at 800 MHz. However, since our radix sort implementation exploits shared memory among all the tiles, it is constrained by the 32-bit addressing space (4 GB). The compiler is a specific version for the Tilera architecture of GCC version *4.4.3*, provided with the *Tilera MDE 3.0.1* IDE. The code has been compiled with the *-O3* optimization flag. We configured the system to expose 62 tiles at user level, while the other 2 are sequestered by the operating systems for executing the drivers of the communication ports. Experiments are executed with one thread per tile.

We used six datasets with varying bit entropy. Lower entropy means higher probability to have similar numbers in the dataset. Dataset 1 is a uniform distribution of pseudo-random numbers generated with the *random_r* POSIX function. Datasets from 2 to 5 are generated by executing multiple bitwise AND operations among uniformly distributed random numbers. Finally, dataset 6 only contains keys with the same value (0). Hence, its entropy is zero. This is equivalent to ANDing all the possible random numbers in the considered range. Table 7.2 reports how many numbers have been used in the AND operation and the entropy of each dataset. The experiments use datasets

Table 7.2: Datasets description.

dataset	keys in *AND*	entropy
1	1	32.00 bit
2	2	25.95 bit
3	3	17.41 bit
4	4	10.78 bit
5	5	6.42 bit
6	inf	0.00 bit

with up to 240 million keys (MK)[1]. We repeated each experiment 10 times, and report the arithmetic mean of the execution times.

7.7.2 Optimization effects

Figure 7.8 reports the throughput (millions of keys sorted per second - MK/sec) of the optimized and non-optimized version of radix sort on the TILEPro64 processor on dataset 1. The non-optimized version is the basic algorithm while the optimized version includes all the optimizations previously discussed. As Figure 7.8 shows, optimizations significantly improve performance and scalability of the algorithm. The non-optimized algorithm with 62 threads shows a performance degradation similar to the *write-rand hash* benchmark of Figure 7.3. The reason is that the performance of the algorithm is dominated by the key distribution phase, and for the non-optimized version this is analogous to the random write operation described in Section 7.3.2.

7.7.3 Scaling

We initially evaluated the scaling of the algorithm by progressively increasing the number of threads (threads are pinned to tiles), while sorting datasets of fixed size (240 MK).

Figure 7.9 shows the throughput of the algorithm on the six datasets with memory allocations striped across memory controllers. Our implementation has an average throughput of 132 MK/sec with 62 threads on dataset 1. The algorithm scales almost linearly up to 32 threads. Over 32 threads the slope decreases, due to higher contention on the memory controllers. Variability for different entropies is in the range of 10%.

Figure 7.10 shows the same set of experiments with allocations on specific memory controllers. For these experiments, memory is allocated through the nearest memory

[1]To obtain adjacent block allocations each block is a multiple of the page size (64 KB), thus the exact number of keys is $243,793,920$

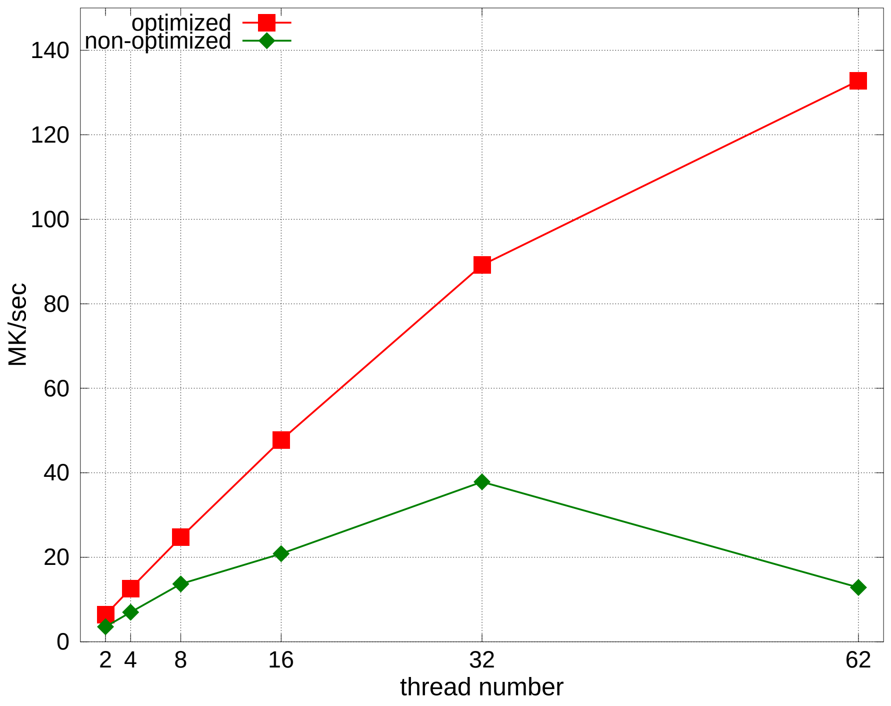


Figure 7.8: Throughput with and without optimizations for the TILEPro64 processor.

controller to the tile that performed the allocation, reducing access latencies for private and local data. In contrast to Figure 7.9, there is a slight change of slope at 16 threads, while the performance keeps scaling linearly when increasing the number of threads from 32 to 62. However, the throughput at 32 threads is lower. Since threads are pinned on tiles in a ordered way (i.e., progressively filling rows), with 32 threads only two (out of four) memory controllers are fully utilized. With striped memory allocations, instead, 32 tiles can exploit all the memory controllers. The peak performance with 62 threads in the two cases is similar, since in average all the four memory controllers are equally utilized. However, access patterns to the NoCs and to the memory controllers change, leading to slightly different behaviors when changing the datasets.

Figure 7.11 shows the throughput of the algorithm when changing dataset sizes. As expected, the throughput increases with larger datasets as the utilization of the processor increases. Performance saturates in the range of 180-210 MK and is mainly bounded by the memory bandwidth for scattered writes, as discussed in Section 7.6.3.

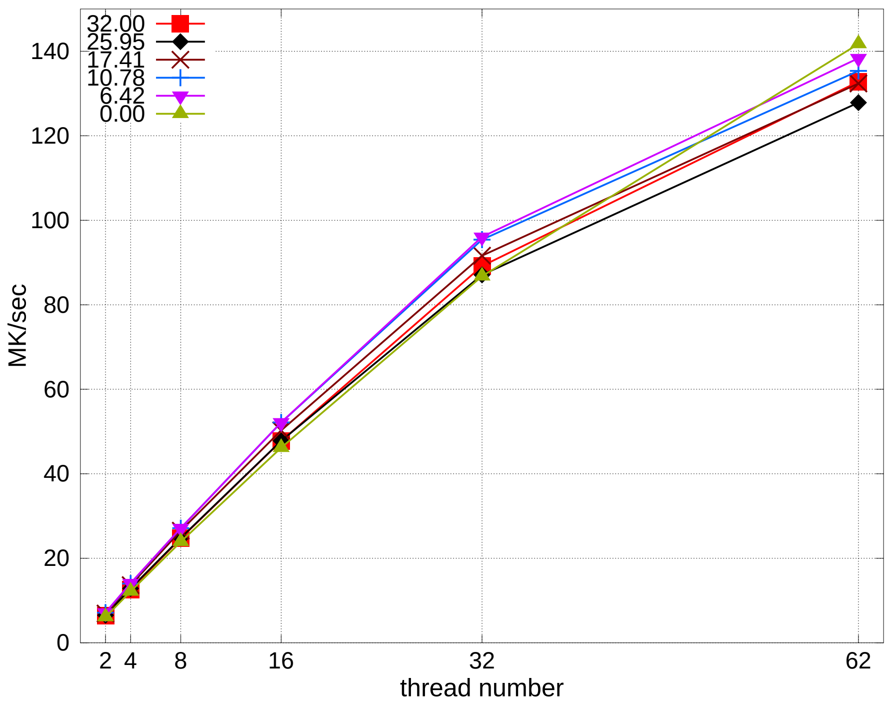


Figure 7.9: Throughput with varying number of threads, 240 MK and striped allocations.

	GPU		x86	Tilera
Platform	Tesla C2070	Tesla C2070 + comm.	Xeon W5590	TILEPro64
Throughput [MK/sec]	664	254	212	132
TDP [Watt]	238	238	130	50.6
Efficiency [MK/sec · Watt]	2.79	1.07	1.63	2.61

Table 7.3: Comparison of various radix sort implementation in terms of throughput (on 32-bit keys), Thermal Design Power (TDP) and performance-per-watt efficiency. For the GPU, we show performance with and without communication of data across the PCI Express bus.

7.7.4 Comparison

Table 7.3 compares the performance of the radix sort implementation for the TILEPro64 processor presented in this work to implementations for Graphic Processing Units (GPUs) and x86 processors. Since the performance of a sorting algorithm depends on the datasets on which it is applied, rather than considering only the results proposed in literature, we chose to use accessible existing implementations or reimplement the proposed approaches. For the GPU, we used the radix sort implementation provided in the CUDA Thrust Libraries, which corresponds to [117], and executed it on a NVIDIA Tesla C2070 board (448 streaming processors at 1.15 GHz, 6 GB of GDDR5 at

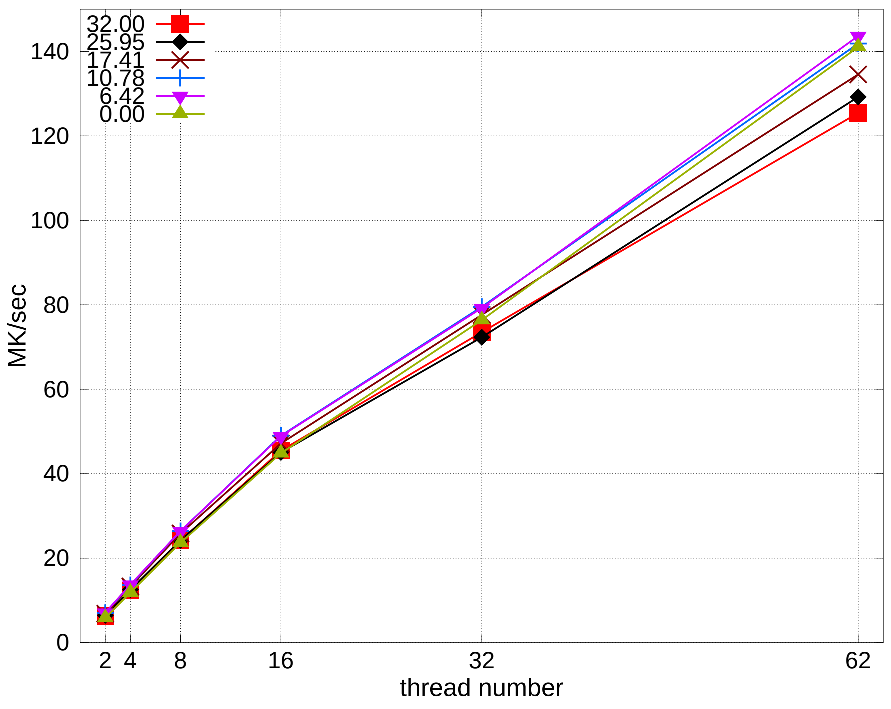


Figure 7.10: Throughput with varying number of threads, 240 MK and allocations on specific memory controllers.

3 GHZ) with CUDA 4.1. For the x86 processor, instead, we re-implemented the cache optimized radix sort algorithm presented in [144], compiled it with GCC 4.6 at -O3 optimization level, and executed it on a system with an Intel Xeon W5590 processor (4 cores with 2 threads each, 256 KB L2 cache per core and unified 8 MB L3 cache, 3.3 GHz) and 32 GB of DDR3-1066 memory. For all the experiments, we used dataset 1. The table also shows the maximum Thermal Design Power (TDP) in Watts and the performance per Watt for each platform. TDPs have been taken from the board specifications [8] for the Tesla C2070, from the Intel website [4] for the Xeon W5590 and from the TILEmpower user guide for the TILEPro64. Although the TDP only represents a possible maximum power dissipation, and actual averages may be lower, we believe that it is a reasonable approximation to understand performance-per-watt behavior without requiring a non-trivial power instrumentation of all the platforms [122].

From the Table, we can see that the GPU implementation reaches the highest performance, with 664 MK/sec. It also appears the most efficient in terms of performance-per-watt. However, when data communication across the PCI Express is considered, the overall performance significantly decreases. Due to the high TDP of the board,

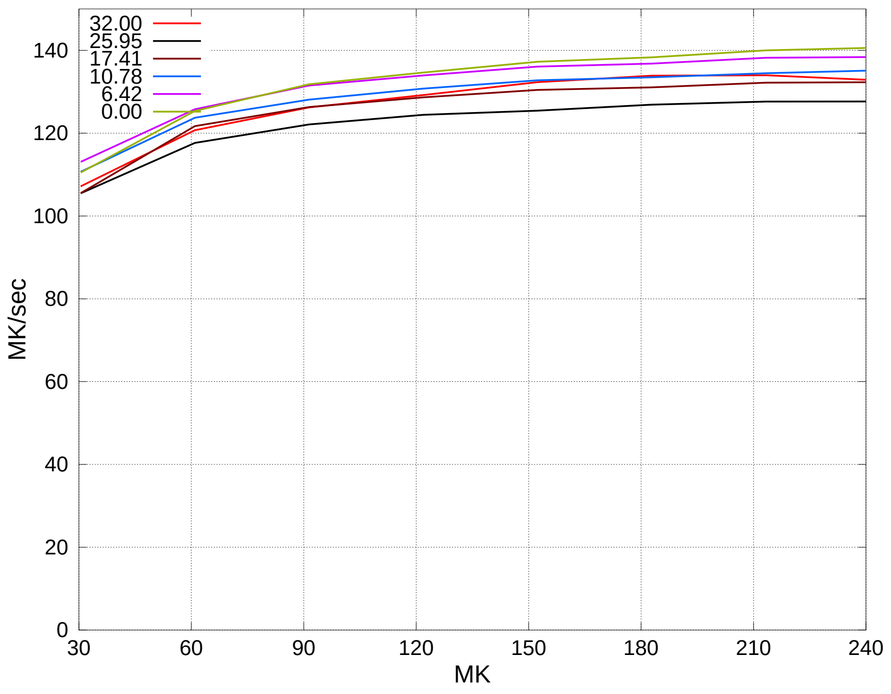


Figure 7.11: Throughput with varying dataset sizes and 62 threads.

when communication is included, the performance-per-watt also becomes the worst of the evaluated platforms. Consequently, the preferred approach should be to use GPU sorting as a phase of more complex algorithms that execute entirely on the GPU. The x86 implementation is slower than the GPU implementation, even considering data transfers. However, the TDP of the Xeon is lower than the TDP of the Tesla C2070, thus performance-per-watt is higher than the GPU implementation with communications. The TILEPro64 implementation has the lowest performance of the solutions considered. However, its TDP is also, by a large margin, the lowest of the three platforms. This leads to a performance-per-watt comparable to the GPU implementation without communication, and higher than the others.

7.8 Conclusions

In this chapter we show how to exploit hardware features of a many-core processor such as Tilera TILEPro64 processor, to improve the performance of a fundamental search algorithm such as the radix sort. The Tilera TILEPro64 processor is one of the first successful low-power manycore architectures. Our implementation employs several optimization techniques that exploit processor's features such as the NoCs and the configurable Dynamic Distributed Cache. We have shown how the optimizations impact the performance of each phase of the algorithm (i.e., local histogram generation, offset computation and keys distribution). We discussed the scalability of the algorithm with respect to the number of tiles (cores) used and the datasets size. We used datasets with varying levels of entropy, showing how keys diversity influences the performance. We compared our optimized radix sort implementation for the TILEPro64 processor with current state-of-the-art implementations for x86 multicores and GPUs, considering throughput and power efficiency. Even if our implementation does not achieve the same peak performance of other architectures, it obtains comparable, or better, results in terms of power efficiency.

The hardware features exploited to improve a specific kernel such as the radix sort, could be integrated into a specialized runtime system to improve the performance metrics of an entire class of applications (i.e. irregular applications).

Chapter 8

Conclusions

The last two decades witnessed an exponential growth of supercomputing performance. HPC system software has evolved to keep pace with technology and applications evolution. Research on scalable system software for large-scale system has historically been driven by two fundamental aspects: on one side, performance, efficiency and reliability, on the other side the goal of providing a productive programming environment.

Exascale systems will be the result of current technology trends meeting with application requirements, and leveraging the lessons learned from more than two decades of supercomputing. The trends in hardware technologies will introduce additional complexity and the need for highly adaptable and scalable system software. Moreover, the difficulty of developing large-scale applications with an explicit message passing programming model as it is commonly done today will introduce newer programming models. These new developments imply radical changes at all levels of the system software stack and in particular for the OS and the runtime system.

The contributions of this thesis are the following:

- Designed and implemented a methodology to provide detailed measurement of system software interruptions and their effect on applications performance.
- Evaluated the effect of TLB misses on applications performance and artificially simulated the effect of complex memory mapping schemas.
- Designed and implemented a runtime system for the class of irregular applications that shows performance improvement of several orders of magnitude with respect to more traditional approaches.
- Implemented several applications optimizations leveraging architecture-specific hardware features and obtaining significant performance improvements. These

optimizations can be integrated into the specialized runtime system to be used by an entire class of applications.

The work described in this thesis is being extended in various directions. A methodology for the detailed measurement of OS noise is being extended to a full large scale system, and also integrated into the runtime system. Given the complexity of new-generation runtime systems, detailed measurement of the runtime system events and communications patterns is needed. The ability to measure and aggregate performance profiling becomes fundamental as the system scale increases. Part of the future work is to integrate performance profiling into the runtime system. Therefore, the runtime system could automatically adapt to the changing workload using information gathered during online performance profiling. The runtime system can integrate information about the specific computation pattern of applications with the performance profiling provided by the OS and implement effective policies to improve efficiency, scalability and reliability.

The specialized runtime system developed (GMT) is being actively extended and tested in order to improve its performance and reliability. GMT is the candidate runtime system to support several new-generation applications such as analytics, big data and in general irregular applications. A large project is using GMT to develop a Semantic Graph Database. In this context, GMT is being extended to provide additional features. New features include the ability of handling parallel IO from global arrays; additional primitives to initialize arrays with a given value and to move large chunks of global memory from one position to another. GMT performance is also being improved redesigning and profiling the context switching and the aggregation mechanism.

GMT is currently available only for x86 architecture, but an adapted version of GMT is being developed to execute on the Tilera many-core processor, to exploits fast core-to-core communications and fine-grain locality control. Plans are also in place to port GMT on the IBM POWER architecture to exploit hardware multithreading and other hardware features specific to the POWER architecture.

Pubblications

Conference papers:

- A Quantitative Analysis of Operating System Noise - Alessandro Morari, Roberto Gioiosa, Robert Wisniewski, Francisco J. Cazorla, Mateo Valero. May 2011 , Anchorage, Alaska, USA. In 25th **IEEE International Parallel & Distributed Processing Symposium (IPDPS).** 2011.

- Evaluating the Impact of TLB Misses on Future HPC Systems - Alessandro Morari, Roberto Gioiosa, Robert W. Wisniewski, Bryan S. Rosenburg, Todd Inglett, Mateo Valero. In 25th **IEEE International Parallel & Distributed Processing Symposium (IPDPS).** 2012.
  This paper won the **IPDPS Best Paper Award**.

- Efficient Sorting on the Tilera Manycore Architecture - Alessandro Morari, Antonino Tumeo, Oreste Villa, Simone Secchi, Mateo Valero. In 24th **International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD).** 2012.

- Accelerating semantic graph databases on commodity clusters - Alessandro Morari, Vito Giovanni Castellana, Oreste Villa, David Haglin, John Feo, Jesse Weaver, and Antonino Tumeo. In **IEEE International Conference on Big Data (IEEE BigData 2013).** 2013.

- Scaling Irregular Applications through Data Aggregation and Software Multithreading - Alessandro Morari, Oreste Villa, Antonino Tumeo, Daniel Chavarría-Miranda, Mateo Valero. In **IEEE International Parallel & Distributed Processing Symposium (IPDPS).** 2014.

Journal articles:

- SMT Malleability in IBM POWER5 and POWER6 Processors - Alessandro Morari, Carlos Boneti, Francisco J. Cazorla, Roberto Gioiosa, Chen-Yong Cher, Alper Buyuktosunoglu, Pradip Bose and Mateo Valero. In **IEEE Transactions on Computers.** 2012.

Workshop papers:

- Toward a Data Scalable Solution for Facilitating Discovery of Scientific Data Resources - Alan Chappell, Sutanay Choudhury, John Feo, David Haglin, Alessandro Morari, Sumit Purohit, Karen Schuchardt, Antonino Tumeo, Jesse Weaver, and Oreste Villa. In **Workshop on Data-Intensive Scalable Computing Systems (DISCS)** held in conjuction with SC13: The International Conference for High Performance Computing, Networking, Storage and Analysis. 2013.

Extended abstracts:

- Analyzing OS noise for HPC systems - Alessandro Morari, Francesco Piermaria, Emiliano Betti, Marco Cesati, Roberto Gioiosa,Francisco J. Cazorla. In 6th **International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES).** 2010.

- HPC System Software for Regular and Irregular Parallel Applications - Alessandro Morari, Mateo Valero. In 26th **IEEE International Parallel & Distributed Processing Symposium (IPDPS).** 2013.

References

[1] http://www.top500.org/lists/2007/06. 100

[2] AMBER: Assisted model building with energy refinement. http://ambermd.org/. 32

[3] Apache Giraph. `http://incubator.apache.org/giraph/`. 93

[4] Intel Xeon Processor W5590. Available at http://ark.intel.com/products/41643/. 170

[5] NAMD scalable molecular dynamics. http://www.ks.uiuc.edu/Research/namd/. 32

[6] OSU Micro-Benchmarks. `http://mvapich.cse.ohio-state.edu/benchmarks/`. 102

[7] TCP BENCHMARK H. Available at http://www.tpc.org/tpch/spec/tpch2.15.0.pdf. 15

[8] Tesla C2050 and Tesla C2070 computing processor board. Board specification. Available at http://www.nvidia.com. 170

[9] Tilera TILEPro64. http://www.tilera.com/products/processors/TILEPRO64. 152

[10] TOP500 - top500 supercomputers list. `http://www.top500.org/system/177790`. xi, 3, 9, 10, 19

[11] Tilera Corporation. Tilera and Quanta unveil the world's most power efficient and highest compute density server. http://www.tilera.com/about_tilera/press-releases/tilera-and-quanta-unveil-worlds-most-power-efficient-and-highest-compute, June 2010. 152

[12] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. S. Vetter, P. Worley, and W. Yu. Early evaluation of ibm bluegene/p. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Piscataway, NJ, USA, 2008. IEEE Press. 49

[13] Saman Amarasinghe, Mary Hall, Pat McCormick, Richard Murphy, Keshav Pingali, Dan Quinlan, Vivek Sarkar, and John Shalf. Exascale programming challenge workshop. Doe ascr, July 2011. 5

[14] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IV, pages 108–120, New York, NY, USA, 1991. ACM. 62

[15] Wendell Anderson, Preston Briggs, C. Stephen Hellberg, Daryl W. Hess, Alexei Khokhlov, Marco Lanzagorta, and Robert Rosenberg. Early Experience with Scientific Programs on the Cray MTA-2. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 46–, New York, NY, USA, 2003. ACM. 90

[16] Jonathan Appavoo, Kevin Hui, Michael Stumm, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Craig A. N. Soules. An infrastructure for multiprocessor run-time adaptation. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 3–8, New York, NY, USA, 2002. ACM. 35

[17] Edoardo Aprà, Alistair P. Rendell, Robert J. Harrison, Vinod Tipparaju, Wibe A. deJong, and Sotiris S. Xantheas. Liquid water: obtaining the right answer for the right reasons. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–7, New York, NY, USA, 2009. ACM. 32

[18] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical report, NASA Advanced Supercomputing (NAS) Division, March 1994. 144

[19] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. The influence of operating systems on the performance of collective operations at extreme scale. *Cluster Computing, IEEE International Conference on*, 0:1–12, 2006. 36

[20] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. Operating system issues for petascale systems. *SIGOPS Oper. Syst. Rev.*, 40(2):29–33, 2006. 36

[21] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, Susan Coghlan, and Aroon Nataraj. Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, 11(1):3–16, 2008. 36

[22] P.H. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. The influence of operating systems on the performance of collective operations at extreme scale. In *Proc. of the 2006 IEEE Int. Conf. on Cluster Computing*, Barcelona, Spain, 2006. 35

[23] C. Bekas, A. Curioni, and I. Fedulova. Low cost high performance uncertainty quantification. In *WHPCF '09: Proceedings of the 2nd Workshop on High Performance Computational Finance*, pages 1–8, New York, NY, USA, 2009. ACM. 32, 59

[24] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 2008. xi, 11, 13, 59

[25] J. Y. Berthou. Final report on roadmap and recommendations development. Technical report, European Exascale Software Initiative, 2011. 3, 12, 13

[26] Emiliano Betti, Marco Cesati, Roberto Gioiosa, and Francesco Piermaria. A global operating system for HPC clusters. In *CLUSTER*, pages 1–10, 2009. 38

[27] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level tlbs for chip multiprocessors. In *HPCA*, pages 62–63. IEEE Computer Society, 2011. 62

[28] Abhishek Bhattacharjee and Margaret Martonosi. Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 29–40, Washington, DC, USA, 2009. IEEE Computer Society. 62

[29] Abhishek Bhattacharjee and Margaret Martonosi. Inter-core cooperative tlb for chip multiprocessors. *SIGARCH Comput. Archit. News*, 38:359–370, March 2010. 62, 80

[30] Guy E. Blelloch. *Vector models for data-parallel computing.* MIT Press, Cambridge, MA, USA, 1990. 152, 158, 163

[31] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995. 24

[32] Dan Bonachea. Gasnet specification, v1.1 - t.r. csd-02-1207. Technical report, UC Berkeley, October 2002. 91, 101

[33] C. Boneti, F. Cazorla, R. Gioiosa, C-Y. Cher, A. Buyuktosunoglu, and M. Valero. Software-Controlled Priority Characterization of POWER5 Processor. In *The 35th Int. Symp. on Computer Architecture (ISCA).*, Beijing, China, June 2008. 122, 124

[34] C. Boneti, R. Gioiosa, F. J. Cazorla, and M. Valero. A dynamic scheduler for balancing HPC applications. In *SC '08: Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, 2008. 49, 122, 124, 131, 142

[35] Carlos Boneti, Roberto Gioiosa, Francisco J. Cazorla, Julita Corbalán, Jesús Labarta, and Mateo Valero. Balancing HPC applications through smart allocation of resources in MT processors. In *Proceedings of the 22th international conference on Parallel and distributed processing*, IPDPS '08, pages 1–12, Miami, Florida, USA, 2008. 124

[36] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel.* O'Reilly, 3rd edition, 2005. 38, 49, 51, 69

[37] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M Badia, Eduard Ayguade, and Jesús Labarta. Productive cluster programming with ompss. In *Euro-Par 2011 Parallel Processing*, pages 555–566. Springer, 2011. 25

[38] David R Butenhof. *Programming with POSIX threads.* Addison-Wesley Professional, 1997. 25

[39] Laura Carrington, Dimitri Komatitsch, Michael Laurenzano, Mustafa M. Tikir, David Michea, Nicolas Le Goff, Allan Snavely, and Jeroen Tromp. High-frequency simulations of global seismic wave propagation using SPECFEM3D_GLOBE on 62k processors. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Piscataway, NJ, USA, 2008. IEEE Press. 32

[40] Umit V. Catalyurek, Florin Dobrian, Assefaw Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. Distributed-memory parallel algorithms for matching and coloring. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1971–1980, Washington, DC, USA, 2011. IEEE Computer Society. 114

[41] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61. ACM, 2011. 24

[42] F.J. Cazorla, P.M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE Trans. Comput.*, 55(7):785–799, 2006. 124

[43] Francisco J. Cazorla, Alex Ramirez, Mateo Valero, Peter M. W. Knijnenburg, Rizos Sakellariou, and Enrique Fernández. QoS for High-Performance SMT Processors in Embedded Systems. *IEEE Micro*, 24:24–31, July 2004. 124

[44] Intel Open Source Technology Center. Open community runtime. http://www.kernel.or://01.org/open-community-runtime. 93

[45] Soumen Chakrabarti and Katherine Yelick. Implementing an irregular application on a distributed memory multiprocessor. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 169–178, New York, NY, USA, 1993. ACM. 90

[46] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007. 24, 91

[47] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM. 26, 91

[48] J. Chen, A. Choudhary, S. Feldman, B. Hendrickson, C.R. Johnson, R. Mount, V. Sarkar, V. White, and D. Williams. Synergistic challenges in data-intensive

science and exascale computing. Doe ascac data subcommittee report, March 2013. 3, 14, 16, 85

[49] Guojing Cong, Gheorghe Almasi, and Vijay Saraswat. Fast PGAS connected components algorithms. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 13:1–13:6, New York, NY, USA, 2009. ACM. 91

[50] Rich Cook, Evi Dube, Ian Lee, Lee Nau, Charles Shered, and Felix Wang. Survey of novel programming models for parallelizing applications at exascale. Technical report, Lawrence Livermore National Laboratory, 2011. 24

[51] Inc. Cray. Urika Big Data Graph Appliance. http://www.cray.com/Products/BigData/uRiKA.aspx, April 2013. 90

[52] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *J. Parallel Distrib. Comput.*, 22(3):462–478, September 1994. 91

[53] P. De, R. Kothari, and V. Mann. Identifying sources of operating system jitter through fine-grained kernel instrumentation. In *Proc. of the 2007 IEEE Int. Conf. on Cluster Computing*, Austin, Texas, 2007. 34, 35

[54] Pradipta De, Vijay Mann, and Umang Mittaly. Handling OS jitter on multicore multithreaded systems. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society. 36

[55] Mathieu Desnoyers and Michel R. Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *Proceedings of the 2006 Linux Symposium*, 2006. 33, 39

[56] Mathieu Desnoyers and Michel R. Dagenais. LTTng, filling the gap between kernel instrumentation and a widely usable kernel tracer. In *Linux Foundation Collaboration Summit 2009 (LFCS 2009)*, April 2009. 33

[57] Matthew DeVuyst, Rakesh Kumar, and Dean M. Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 140–140, Washington, DC, USA, 2006. 122, 124

[58] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011. 6, 59

[59] Gautham K. Dorai and Donald Yeung. Transparent Threads: Resource Sharing in SMT Processors for High Single-Thread Performance. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 30–, Washington, DC, USA, 2002. 146

[60] Tarek El-Ghazawi and Lauren Smith. Upc: unified parallel c. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 27. ACM, 2006. 26, 60, 91

[61] Dawson R. Engler and M. Frans Kaashoek. Exterminate all operating system abstractions. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 78–83, Orcas Island, Washington, May 1995. IEEE Computer Society. 35

[62] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995. 35

[63] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole, Jr. The operating system kernel as a secure programmable machine. *Operating Systems Review*, 29(1):78–82, January 1995. 35

[64] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376, 2011. 12

[65] Stijn Eyerman and Lieven Eeckhout. Per-thread cycle accounting in smt processors. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 133–144, New York, NY, USA, 2009. 124

[66] John Feo, David Harper, Simon Kahan, and Petr Konecny. ELDORADO. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, New York, NY, USA, 2005. ACM Press. 86, 90

[67] John Feo, Oreste Villa, Antonino Tumeo, and Simone Secchi. Irregular applications: architectures &#38; algorithms. In *Proceedings of the first workshop on Irregular applications: architectures and algorithm*, IAAA '11, pages 1–2, New York, NY, USA, 2011. ACM. 4

[68] Kurt B. Ferreira., Patrick G. Bridges, and Ron Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press. 31, 34, 35, 60, 71, 75, 77, 80

[69] MPI Forum. MPI: A message passing interface standard. 8, 1994. 32

[70] G.R. Gao, T. Sterling, R. Stevens, M. Hereld, and Weirong Zhu. Parallex: A study of a new parallel computation model. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–6, 2007. 25, 92

[71] M. Giampapa, T. Gooding, T. Inglett, and R. W. Wisniewski. Experiences with a lightweight supercomputer kernel: Lessons learned from blue gene's cnk. 2010. 32, 35, 36, 46

[72] M. E. Giampapa, R. Bellofatto, M. A. Blumrich, D. Chen, M. B. Dombrowa, A. Gara, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, B. J. Nathanson, B. D. Steinmacher-Burow, M. Ohmacht, V. Salapura, and P. Vranas. Blue Gene/L advanced diagnostics environment. *IBM Journal for Research and Development*, 2005. 60

[73] B. Gibbs, B. Atyam, F. Berres, B. Blanchard, L. Castillo, P. Coelho, N. Guerin, L. Liu, C. Diniz Maciel, and C. Thirumalai. *Advanced POWER Virtualization on IBM eServer p5 Servers: Architecture and Performance Considerations*. IBM Redbook, 2005. 128

[74] R. Gioiosa, S. McKee, and M. Valero. Designing os for hpc applications: Scheduling. *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2010)*, September 2010. 35, 36, 49

[75] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare. Analysis of system overhead on parallel computers. In *The 4th IEEE Int. Symp. on Signal Processing and Information Technology (ISSPIT 2004)*, Rome, Italy, December 2004. http://bravo.ce.uniroma2.it/home/gioiosa/pub/isspit04.pdf. 31, 34, 35, 38, 46, 52, 60, 75, 77, 80

[76] R. Giorgi, C.A. Prete, G. Prina, and L. Ricciardi. Trace factory: generating workloads for trace-driven simulation of shared-bus multiprocessors. *Concurrency, IEEE*, 5(4):54–68, Oct 1997. 35

[77] Roberto Giorgi. Teraflux: exploiting dataflow parallelism in teradevices. In *Conf. Computing Frontiers*, pages 303–304, 2012. 27

[78] GNU. Gnu libc manual. http://www.gnu.org/software/libc/manual/html_node/System-V-contexts.html. 105

[79] David B. Golub, Daniel P. Julin, Richard F. Rashid, Richard P. Draves, Randall W. Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. Microkernel operating system architecture and mach. In *In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, 1992. 19

[80] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association. 86

[81] The graph 500 list. `http://www.graph500.org`, April 2013. 110

[82] Fei Guo, Yan Solihin, Li Zhao, and Ravishankar Iyer. A framework for providing quality of service in chip multi-processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 343–355, Washington, DC, USA, 2007. 124

[83] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010. 60

[84] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient gather and scatter operations on graphics processors. In *SC '07: ACM/IEEE conference on Supercomputing*, pages 46:1–46:12, 2007. 158

[85] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2011. 16

[86] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006. 15, 123, 144

[87] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993. 60

[88] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986. 163

[89] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Nov. 2010. 5

[90] Jerry Huck and Jim Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 39–50, New York, NY, USA, 1993. ACM. 62

[91] IBM. PowerPC Architecture book: Book III: PowerPC Operating Environment Architecture. 128

[92] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. *SIGMETRICS Perform. Eval. Rev.*, 35:25–36, June 2007. 124

[93] Guohua Jin, J. Mellor-Crummey, L. Adhianto, W.N. Scherer, and Chaoran Yang. Implementation and Performance Evaluation of the HPC Challenge Benchmarks in Coarray Fortran 2.0. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1089 –1100, May 2011. 91

[94] H. Jin and R.F. Van der Wijngaart. Performance characteristics of the multi-zone nas parallel benchmarks. *J. Parallel Distrib. Comput.*, 66(5):674–685, 2006. 145

[95] Terry Jones, Shawn Dawson, Rob Neel, William Tuel, Larry Brenner, Jeffrey Fier, Robert Blackmore, Patrick Caffrey, Brian Maskell, Paul Tomlinson, and Mark Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, page 10, Washington, DC, USA, 2003. IEEE Computer Society. 35, 36

[96] Laxmikant V Kale and Sanjeev Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993. 24, 60, 92

[97] R. Kalla, B. Sinharoy, and J.M. Tendler. Ibm power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24:40–47, 2004. 122, 139

[98] Gokul B. Kandiraju and Anand Sivasubramaniam. Characterizing the d-tlb behavior of spec cpu2000 benchmarks. In *In Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 129–139. ACM Press, 2002. 62

[99] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on parallel programming model. In *Proceedings of the IFIP International Conference on Network and Parallel Computing*, NPC '08, pages 266–275, Berlin, Heidelberg, 2008. Springer-Verlag. 22

[100] Darren J. Kerbyson, Philip W. Jones, Darren J. Kerbyson, and Philip W. Jones. A performance model of the parallel ocean program. *International Journal of High Performance Computing Applications*, 19, 2005. 32, 59

[101] Khronos Group. *The OpenCL Specification*, September 2010. 25

[102] Rob Knauerhase, Romain Cledat, and Justin Teller. For extreme parallelism, your os is sooooo last-millennium. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, HotPar'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association. 4

[103] Jesus Labarta. Starss: A programming model for the multicore era. In *PRACE WorkshopNew Languages & Future Technology Prototypes at the Leibniz Supercomputing Centre in Garching (Germany)*, 2010. 25

[104] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, April 2010. 32, 33, 60

[105] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM J. Res. Dev.*, 51:639–662, November 2007. 122, 125, 126, 140

[106] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society. 15

[107] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with MapReduce: a survey. *SIGMOD Rec.*, 40(4):11–20, January 2012. 93

[108] Shin-Jae Lee, Minsoo Jeon, Dongseung Kim, and Andrew Sohn. Partitioned parallel radix sort. *J. Parallel Distrib. Comput.*, 62:656–668, April 2002. 158

[109] Jochen Liedtke. Improving IPC by kernel design. In *SOSP '93: Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 175–188, New York, NY, USA, 1993. ACM. 35

[110] LLNL. Sequoia benchmarks. https://asc.llnl.gov/sequoia/benchmarks/. 18, 32, 34, 43, 59, 61

[111] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012. 86, 93

[112] Carlos Luque, Miquel Moreto, Francisco J. Cazorla, Roberto Gioiosa, Alper Buyuktosunoglu, and Mateo Valero. Cpu accounting in cmp processors. *IEEE Comput. Archit. Lett.*, 8:17–20, January 2009. 124

[113] Carlos Luque, Miquel Moreto, Francisco J. Cazorla, Roberto Gioiosa, Alper Buyuktosunoglu, and Mateo Valero. Itca: Inter-task conflict-aware cpu accounting for cmps. In *Proceedings of the 2009 International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 203–213, Washington, DC, USA, 2009. 124

[114] Zhiqiang Ma and Lin Gu. The Limitation of MapReduce: A Probing Case and a Lightweight Solution. In *CLOUD COMPUTING 2010: the 1st Intl. Conf. on Cloud Computing, GRIDs, and Virtualization*, 2010. 93

[115] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD '10: ACM International Conference on Management of data*, pages 135–146, 2010. 93

[116] Collin McCurdy, Alan L. Cox, and Jeffrey Vetter. Investigating the tlb behavior of high-end scientific applications on commodity microprocessors. In *Proceedings of the ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software*, pages 95–104, Washington, DC, USA, 2008. IEEE Computer Society. 62

[117] Duane Merrill and Andrew Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(02):245–272, 2011. 158, 169

[118] Ingo Molnr. [patch] Modular Scheduler Core and Completely Fair Scheduler [CFS] linux-kernel mailing list., 2007. http://lwn.net/Articles/230501/. 48

[119] José E. Moreira, Michael Brutman, nos José Casta Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Rober Haskin, Todd Inglett, Derek Lieber, Pat McCarthy, Mike Mundy, Jeff Parker, and Brian Wallenfelt. Designing a highly-scalable operating system: the Blue Gene/L story. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 118, New York, NY, USA, 2006. ACM. 35, 36, 60

[120] Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Rizos Sakellariou, and Mateo Valero. FlexDCP: a QoS framework for CMP architectures. *SIGOPS Oper. Syst. Rev.*, 43:86–96, April 2009. 125

[121] MPI forum. The Message Passing Interface (MPI) standard. http://www.mpi-forum.org/. 24, 32

[122] A. Munir, S. Ranka, and A. Gordon-Ross. High-performance energy-efficient multicore embedded computing. *Parallel and Distributed Systems, IEEE Transactions on*, 23(4):684 –700, april 2012. 170

[123] Jens Muttersbach, Thomas Villiger, and Wolfgang Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Advanced Research in Asynchronous Circuits and Systems, 2000.(ASYNC 2000) Proceedings. Sixth International Symposium on*, pages 52–59. IEEE, 2000. 26

[124] NASA. NAS parallel benchmarks. http://www.nas.nasa.gov/Resources /Software/npb.html. 32

[125] Aroon Nataraj and Matthew Sottile. The ghost in the machine: Observing the effects of kernel operation on parallel application performance. In *SC'7: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007. 34

[126] Jacob Nelson, Brandon Myers, A. H. Hunter, Preston Briggs, Luis Ceze, Carl Ebeling, Dan Grossman, Simon Kahan, and Mark Oskin. Crunching large graphs with commodity processors. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, HotPar'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association. 92

[127] Kyle J. Nesbit, James Laudon, and James E. Smith. Virtual private caches. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 57–68, New York, NY, USA, 2007. 124

[128] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High Performance Remote Memory Access Communication: The ARMCI Approach. *Int. J. High Perform. Comput. Appl.*, 20(2):233–253, 2006. 60, 91

[129] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, May 2006. 24

[130] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. Advances, Applications and Performance of the

Global Arrays Shared Memory Programming Toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, 2006. 91

[131] Robert W Numrich and John Reid. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998. 24

[132] Robert W. Numrich and John Reid. Co-arrays in the next Fortran Standard. *SIGPLAN Fortran Forum*, 24(2):4–17, August 2005. 91

[133] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008. 23

[134] OpenMP Architecture Review Board. The OpenMP specification for parallel programming. Available at http://www.openmp.org. 25, 32, 60

[135] Yoonho Park, Eric Van Hensbergen, Marius Hillenbrand, Todd Inglett, Bryan Rosenburg, Kyung Dong Ryu, and Robert Wisniewski. Poster: Fusedos: A hybrid approach to exascale operating systems. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 1417–, Washington, DC, USA, 2012. IEEE Computer Society. 5

[136] Krzysztof Parzyszek, Jarek Nieplocha, and Ricky A Kendall. A generalized portable shmem library for high performance computing. Technical report, Ames Lab., Ames, IA (US), 2000. 26

[137] F. Petrini, D. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *ACM/IEEE SC2003*, Phoenix, Arizona, November 10–16, 2003. 4, 31, 34, 35, 46, 60, 75, 77, 80

[138] Vincent Pillet, Vincent Pillet, Jess Labarta, Toni Cortes, Toni Cortes, Sergi Girona, Sergi Girona, and Departament D'arquitectura De Computadors. Paraver: A tool to visualize and analyze parallel code. Technical report, In WoTUG-18, 1995. 33, 38, 40

[139] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Architectural support for operating system-driven cmp cache management. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 2–12, New York, NY, USA, 2006. 124

[140] Rolf Riesen, Ron Brightwell, Patrick G. Bridges, Trammell Hudson, Arthur B. Maccabe, Patrick M. Widener, and Kurt Ferreira. Designing and implementing

lightweight kernels for capability computing. *Concurr. Comput.: Pract. Exper.*, 21(6):793–817, 2009. 4, 32, 35, 60

[141] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The impact of architectural trends on operating system performance. In *In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 285–298, 1995. 62

[142] Vivek Sarkar and John Hennessy. Partitioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 202–211, New York, NY, USA, 1986. ACM. 22

[143] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *IPDPS '09: IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10, 2009. 158

[144] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort. In *SIGMOD '10: ACM SIGMOD International conference on Management of data*, pages 351–362, 2010. 152, 158, 170

[145] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast Sort on CPUs, GPUs and Intel MIC Architectures. Technical report, Intel Labs, 2010. 158

[146] T. Shanley. *InfiniBand Network Architecture*. Mindshare, Inc., 2002. 49

[147] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997. 60

[148] Edi Shmueli, George Almasi, José Brunheroto, nos José Casta Gabor Dozsa, Sameer Kumar, and Derek Lieber. Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/L. In *ICS '08: Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 165–174, New York, NY, USA, 2008. ACM. 5, 31, 36, 60, 63, 65, 74, 77

[149] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005. 122, 125, 126, 134

[150] Allan Snavely, Larry Carter, Jay Boisseau, Amit Majumdar, Kang Su Gatlin, Nick Mitchell, John Feo, and Brian Koblenz. Multi-processor performance on the Tera MTA. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–8, Washington, DC, USA, 1998. IEEE Computer Society. 90

[151] Allan Snavely, Dean M. Tullsen, and Geoff Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '02, pages 66–76, New York, NY, USA, 2002. 124, 129

[152] Andrew Sohn and Yuetsu Kodama. Load balanced parallel radix sort. In *ICS '98: international conference on Supercomputing*, pages 305–312, 1998. 158

[153] Marco Solinas, Rosa M. Badia, Franois Bodin, Albert Cohen, Paraskevas Evripidou, Paolo Faraboschi, Bernhard Fechner, Guang R. Gao, Arne Garbade, Sylvain Girbal, Daniel Goodman, Behran Khan, Souad Koliai, Feng Li, Mikel Lujn, Laurent Morin, Avi Mendelson, Nacho Navarro, Antoniu Pop, Pedro Trancoso, Theo Ungerer, Mateo Valero, Sebastian Weis, Ian Watson, Stphane Zuckermann, and Roberto Giorgi. The teraflux project: Exploiting the dataflow paradigm in next generation teradevices. In *Proceedings of the 2013 Euromicro Conference on Digital System Design*, DSD '13, pages 272–279, Washington, DC, USA, 2013. IEEE Computer Society. 4

[154] M. Sottile and R. Minnich. Analysis of microbenchmarks for performance tuning of clusters. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 371–377, Washington, DC, USA, 2004. IEEE Computer Society. 7, 38, 56

[155] P. Terry, A. Shan, and P. Huttunen. Improving application performance on HPC systems with process synchronization. *Linux Journal*, November 2004. http://www.linuxjournal.com/article/7690. 35

[156] D. Tsafrir, Y. Etsion, D.G. Feitelson, and S. Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *ICS '05: Proc. of the 19th Annual International Conference on Supercomputing*, pages 303–312, New York, NY, USA, 2005. ACM Press. 31, 34, 35, 38, 52

[157] Nathan Tuck and Dean M. Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 26–, Washington, DC, USA, 2003. 124

[158] Dean M. Tullsen and Jeffery A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 318–327, Washington, DC, USA, 2001. 124

[159] Antonino Tumeo, Simone Secchi, and Oreste Villa. Designing Next-Generation massively multithreaded architectures for irregular applications. *Computer*, 45(8):53–61, August 2012. 4

[160] Richard Uhlig, David Nagle, Tim Stanley, Trevor Mudge, Stuart Sechrest, and Richard Brown. Design tradeoffs for software-managed tlbs. *ACM Trans. Comput. Syst.*, 12:175–205, August 1994. 62

[161] UPC Consortium. UPC Language Specifications v. 1.2. `www.gwu.edu/~upc/docs/upc_specs_1.2.pdf`, May 2005. 60

[162] Javier Vera, Francisco J. Cazorla, Alex Pajuelo, Oliverio J. Santana, Enrique Fern, and Mateo Valero. Measuring the performance of multithreaded processors. In *2007 SPEC Benchmark Workshop*, Austin, TX, USA, 2007. 129, 131

[163] Javier Vera, Francisco J. Cazorla, Alex Pajuelo, Oliverio J. Santana, Enrique Fernandez, and Mateo Valero. Fame: Fairly measuring multithreaded architectures. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 305–316, Washington, DC, USA, 2007. 129, 131

[164] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th annual international symposium on Computer architecture*, ISCA '92, pages 256–266, New York, NY, USA, 1992. ACM. 23

[165] Jan Wassenberg and Peter Sanders. Engineering a multi-core radix sort. In *EuroPar '11: international conference on Parallel processing - Part II*, pages 160–169, 2011. 159

[166] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. Active pebbles: parallel programming for data-driven applications. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 235–244, New York, NY, USA, 2011. ACM. 23, 92

[167] Shixiong Xu and Li Chen. Shared work list: hacking amorphous data parallelism in UPC. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '12, pages 124–133, New York, NY, USA, 2012. ACM. 91

[168] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, A. Kamil, K. Datta, P. Colella, and T. Wen. Parallel Languages and Compilers: Perspective from the Titanium Experience. *Int. J. High Perform. Comput. Appl.*, 21(3):266–290, August 2007. 91

[169] Katherine A. Yelick. Programming models for irregular applications. *SIGPLAN Not.*, 28:28–31, January 1993. 86

[170] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, et al. Titanium: A high-performance java dialect. *Concurrency Practice and Experience*, 10(11-13):825–836, 1998. 26