University of Cyprus

DEPARTMENT OF COMPUTER SCIENCE

Improving the Performance of Single and Multi-Application Workloads on Heterogeneous Clustered Many-Core Platforms

Panayiotis Petrides

A Dissertation Submitted to the University of Cyprus in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

May, 2018

VALIDATION PAGE

Doctoral Candidate: Panayiotis Petrides

Doctoral Dissertation Title: Improving the Performance of Single and Multi-Application Workloads on Heterogeneous Clustered Many-Core Platforms

The present Doctoral Dissertation was submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy at the Department of Computer Science and was approved on the ***May 4, 2018*** *by the members of the Examination Committee.*

Examination Committee:

Research Supervisor: ____________________
Associate Professor Pedro Trancoso

Committee Member: ____________________
Professor Constantinos Pattichis

Committee Member: ____________________
Professor Paraskevas Evripidou

Committee Member: ____________________
Professor João Cardoso

Committee Member: ____________________
Professor Dimitris Gizopoulos

DECLARATION OF DOCTORAL CANDIDATE

The present Doctoral Dissertation was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy of the University of Cyprus. It is a product of original work of my own, unless otherwise mentioned through references, notes, or any other statements.

. . . . [Full Name of Doctoral Candidate]

. . . . [Signature of Doctoral Candidate]

Περίληψη

Τα τελευταία χρόνια οι αρχιτεκτονικές επεξεργαστών έχουν αναπτυχθεί προς την κατεύθυνση των πολλαπλών πυρήνων, με αποτέλεσμα την βελτίωση της επίδοσης τους αποφεύγοντας ταυτόχρονα τους περιορισμούς από την κατανάλωση ενέργειας. Ο αυξανόμενος αριθμός πυρήνων σε ένα ολοκληρωμένο κύκλωμα δεν προσφέρει μόνο τα πλεονεκτήματα της δυνητικής μαζικής παραλληλίας αλλά ταυτόχρονα δίνει την δυνατότητα στους κατασκευαστές να εξερευνήσουν νέες αρχιτεκτονικές, όπως την ενσωμάτωση στο ίδιο ολοκληρωμένο κύκλωμα πυρήνων διαφορετικών χαρακτηριστικών. Τα οφέλη αυτών των αρχιτεκτονικών συνοδεύονται όμως και με προκλήσεις.

Ο αυξανόμενος αριθμός πυρήνων σε μια πολυπύρηνη αρχιτεκτονική μπορεί να τύχει εκμετάλλευσης από εφαρμογές με υψηλό βαθμό παραλληλίας. Η μεταφορά μιας εφαρμογής σε αυτού του είδους τις αρχιτεκτονικές δεν είναι μια απλή διαδικασία αλλά μια ευρύτερη εργασία που λαμβάνει υπόψιν τόσο την αρχιτεκτονική του συστήματος όσο και τα χαρακτηριστικά της εφαρμογής. Ως μελέτη περίπτωσης (case study), χρησιμοποιήθηκαν εφαρμογές συστημάτων υποβοήθησης λήψης αποφάσεων (Decision Support System), οι οποίες μεταφέρθηκαν σε αρχιτεκτονική πολλαπλών πυρήνων χρησιμοποιώντας την κοινή ενσωματωμένη στο κύκλωμα μνήμη (on-chip shared memory) για την προεπεξεργασία δεδομένων (prefetching buffer). Τα αποτελέσματα δείχνουν ότι όταν οι αιτήσεις για δεδομένα αντιμετωπίζονται ικανοποιητικά τότε γίνεται εκμεταλλεύσιμη και η παραλληλία των εφαρμογών.

Ενώ κάποιες εφαρμογές επωφελούνται από τον αυξανόμενο αριθμό παράλληλων πυρήνων, σε πολλές περιπτώσεις η χρήση πολυπύρηνων επεξεργαστών στοχεύει στην παράλληλη εκτέλεση πολλαπλών εφαρμογών. Αυτό μπορεί να οδηγήσει σε παρεμβολές μεταξύ των υπό εκτέλεση εφαρμογών. Για την αντιμετώπιση αυτής της πρόκλησης, προτάθηκε μια απλή και μη παρεμβατική προσέγγιση χρησιμοποιώντας τεχνολογία εικονικοποίησης (virtualization techniques) στον ίδιο επεξεργαστή. Οι διαφορετικές εικονικές μηχανές μπορούν να θεωρηθούν ως Τομείς Επίδοσης (Performance Domains) προσφέροντας προβλεψιμότητα επίδοσης για τις διάφορες εφαρμογές. Τα πειραματικά αποτελέσματα δείχνουν ότι επιτυγχάνεται απομόνωση της εκτέλεσης των εφαρμογών σε ένα εικονικοποιημένο περιβάλλον και ταυτόχρονα μειώνονται οι παρεμβολές μεταξύ των εφαρμογών.

Οι μελλοντικοί επεξεργαστές πολλαπλών πυρήνων μεγάλης κλίμακας αναμένεται να είναι μια συλλογή συμπλεγμάτων ετερογενών πυρήνων για να ικανοποιήσουν τις απαιτήσεις των εφαρμογών. Προκειμένου να ικανοποιηθεί η δυναμική συμπεριφορά των εφαρμογών, προτείνεται ένα σύστημα χρόνου εκτέλεσης (run time system) το οποίο είναι υπεύθυνο για την εύρεση ενός καλύτερου πόρου που ταιριάζει σε μια εφαρμογή σε κάθε διαφορετική φάση της εκτέλεσής τους. Ο προτεινόμενος ετερογενής χρονοπρογραμματιστής (scheduler) αξιολογήθηκε τόσο σε πραγματική αρχιτεκτονική πολλαπλών πυρήνων (Intel SCC48 πυρήνων) όσο και με τη χρήση προσομοιωτή (Sniper) για εφαρμογές από τη σουίτα αναφοράς SPEC CPU2006. Τα αποτελέσματα δείχνουν ότι η μεταφορά εφαρμογών σε πυρήνες που ταιριάζουν καλύτερα στις απαιτήσεις τους, οδηγούν σε μείωση του χρόνου εκτέλεσης τους μεταξύ 15% και 36% σε σύγκριση με τυχαίο στατικό χρονοπρογραμματισμό.

Δεδομένης της αυξανόμενης πολυπλοκότητας και πολυμορφίας των πυρήνων του επεξεργαστή, καθώς και των απαιτήσεων της εφαρμογής, θα χρειαστεί η ανάπτυξη περισσότερων από τις προαναφερόμενες τεχνικές για την αντιμετώπιση των προκλήσεων. Επομένως, οι μελλοντικοί ετερογενείς επεξεργαστές πολλαπλών πυρήνων θα πρέπει να περιλαμβάνουν ένα στρώμα εικονικοποίησης το οποίο θα μπορούσε να αποτελείται από όλες τις προτεινόμενες τεχνικές αλλά και άλλες με ένα αρθρωτό τρόπο ούτως ώστε να υποστηρίζει και πυρήνες που αλλάζουν δυναμικά τα χαρακτηριστικά τους.

Abstract

In recent years processor architectures have evolved towards chips with multiple cores, thus delivering the expected performance while avoiding the power wall. Increasing the number of devices on a chip will not only offer the benefit of increasing the potential for parallelism but will also allow manufacturers to explore new designs such as including in the same chip cores of different characteristics. The benefits will come also with challenges in exploiting the performance both for single and multi-application workloads.

The increasing number of cores on a clustered many-core architecture can be exploited by applications with high degree of parallelism. Porting an application for such architectures is not trivial but a joint task of considering both the underlying architecture and the applications' behavior. Memory-bound applications with high degree of parallelism can create an increasing number of memory requests, which must be satisfied without becoming a performance bottleneck. As a case study, a Decision Support System (DSS) workloads was ported to a clustered many-core architecture and the on-chip memory, shared among all cores, was used as a prefetching buffer. The results show that parallelism can be well exploited when the memory requests are well handled.

While some applications benefit from the increasing number of parallel cores, in many cases the use of many-core processors will be for the co-execution of multiple applications. This might happen because of the limited degree of parallelism of the applications or in order to achieve higher throughput and resource utilization. Nevertheless, this can lead to application interference. To address this, a simple

and non-intrusive approach using virtualization on the same processor was proposed. The different Virtual Machines can be seen as Performance Domains since the isolation offers performance predictability for the different applications. The experimental results show that the performance overhead of executing on a virtualized environment is not significant.

While Performance Domains provide isolation, they are static containers that do not adapt well to the dynamic behavior of applications. Future large-scale many-core processors are expected to be organized as a collection of NUMA clusters of heterogeneous cores as to satisfy applications demands. In order to satisfy the applications' dynamic behavior, a runtime system (monitor and scheduler), is proposed. This system is responsible for finding a best matching resource for an application at a certain execution phase. The proposed heterogeneous and NUMA-aware scheduler was evaluated both on a real many-core architecture (48-core Intel SCC) and using a simulator (Sniper) for workloads composed of applications from the SPEC CPU2006 benchmark suite. The results indicate that even when all cores are busy, migrating processes to cores that match better the requirements of applications results in a reduction of the execution time between 15% and 36% compared to a random static scheduling.

Given the increasingly complexity and diversity in the hardware resources, as well as the application demands, more and more of the above-mentioned techniques should be developed to address the challenges. Therefore, the vision is for future heterogeneous many-core processors is to include a virtualization layer which could be composed of all of the proposed techniques and others in a modular way and thus also be able to even support hardware that changes dynamically at runtime.

Acknowledgments

A long journey full of experiences and knowledge has been completed. A journey that provided me with knowledge not only in the field of this thesis, but also knowledge and experience of how to overcome difficulties and how to get the most of each experience. This journey would have never been completed without the continuous support and believe in this work by my advisor, Pedro Trancoso. His contribution and advice were valuable both in terms of completing this thesis but also to help overcoming the difficulties and challenges which this road was full of. Thank you for believing in this journey and holding the compass as my mentor.

The ones I owe the most of appreciation and I am thankful for believing in me are my parents, Petros and Eleni. I thank you for being in my life and providing me with your advices and support. I am grateful of having you in my life and with your own way are supporting me not only during this journey but to my whole life.

Also I wanted to thank my two sisters, Irene and Katerina. With your support and help you helped to this journey each one with your own way. Your help was continuous and valuable to me.

A big thank you to a special person in my heart, Eleni. I could see in your eyes your believe of succeeding through this journey and I can assure you that it was of the greatest help. Thank you for your patience, your support and your love.

I also wanted to thank two little guys. Chuck and Bass. You both changed my world and taught me how to become a better person. I will always be grateful for the time spent together and your unconditional devotion.

Finally, I wanted to thank all the CASPER group members that I have had the pleasure to work together, Kyriakos, Demos, Marios, Andreas and Constantinos. Our collaboration means a lot and my memories from CASPER group will accompany me.

To everybody at the Department of Computer Science, thank you for the excellent cooperation all these years. Special thanks to Melina and Savvoula and to the IT team, Savvas and Maria for providing all the support and even more.

Contributions of this Thesis

Conference and workshop proceedings:

1. **P. Petrides** and P. Trancoso. *"Heterogeneous- and NUMA-aware Scheduling for Many-core Architectures"*, in Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR 2017), pp. 2:1-2:12, Haifa, Israel, May 2017.

2. **P. Petrides** and P. Trancoso. *"Addressing the Challenges of Future Large-Scale Many-core Architectures"*, in Proceedings of the ACM International Conference on Computing Frontiers (CF '13), pp. 1-4, Ischia, Italy, May 2013.

3. **P. Petrides**, A. Diavastos, C. Christofi, and P. Trancoso. *"Scalability and Efficiency of Database Queries on Future Many-Core Systems"*, in Proceedings of the 21st IEEE Euromicro international Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp. 24-28, February 2013.

4. **P. Petrides**, G. Nicolaides and P. Trancoso. *"HPC Performance Domains on Multi-core Processors with Virtualization"*, in Proceedings of the 25th International Conference on Architecture of Computing Systems (ARCS 2012), pp. 123-134, Munich, Germany, February 2012.

5. **P. Petrides**, A. Diavastos, and P. Trancoso. *"Exploring Database Workloads on Future Clustered Many-Core Architectures"*, in Proceedings of the 3rd Many-core Applications Research Community Symposium (MARC 2011), Ettlingen, Germany, July 2011.

6. **P. Petrides**, F. Pratas, L. Sousa, and P. Trancoso, *"Virtualization for Morphable Multi-Cores"*, in Proceedings of the 2nd Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA 2011), Lake Como, Italy, February 2011.

Technical reports:

7. **P. Petrides**, F. Pratas, L. Sousa, P. Trancoso. *"Exploiting Location-Aware Task Execution on Future Large-scale Many-core Architectures"*, University of Cyprus Technical Report TR-12-4, Computer Science Department, on May 24, 2012.

8. **P. Petrides**, F. Pratas, L. Sousa, P. Trancoso, *"Virtualization for Morphable Multi-Cores"*, HiPEAC Technical Report, TR-HiPEAC-0013, July 2010.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Achieving high levels of application performance on a many-core architecture environment considering the characteristics of both available resources and applications demands is not a trivial task. Different approaches exist in order to exploit the increasing number of resources and at the same time target on high performance of applications. In particular, one approach is to target single application performance by exploiting its own parallelism while another approach is to target multiple application performance by exploiting throughput parallelism. Both approaches result in different challenges, which are identified as: *(i)* tuning single application performance by considering both many-core underlying resources and application characteristics, *(ii)* minimizing interference between co-executing applications and *(iii)* satisfying the dynamic demands of applications when executing on a clustered heterogeneous many-core environment.

1.1 Motivation

The current *de-facto* standard in processor design is the multi-core architecture which emerged as a way to provide at the same time energy efficiency and increasing computing power. Moreover, as technology and on-chip integration keeps evolving, the number of cores per chip tends to increase resulting in what is known as many-core architectures. At the same time, processors start integrating cores of different characteristics and include features to change their characteristics dynamically at runtime as to improve their efficiency. While the increasing number of cores may result in a larger degree of parallelism, improving an application performance by exploiting its parallelism should be a joint task of considering both underlying architecture and applications behavior. Additionally, throughput performance of such systems will raise the challenges of co-executing multiple applications of different demands, both in terms of memory and computational resources, within the same chip. Considering processors evolution, they are becoming very complex systems and as the number of cores increases some features make the execution non-uniform across different cores. Cores grouped into clusters are a way to overcome design complexity but in turn it leads to new challenges. The key for determining the best matching core for a certain application is to find out the application's memory and computational requirements. Given that no previous knowledge of the application is assumed, whenever an application enters the system, it is important to have the mechanisms to acquire enough information about its behaviour in terms of resource requirements. A runtime environment, which considers both application demands and available resources, is needed for hiding the underlying architecture to running applications and at the same time to satisfy applications demands and utilize the available resources.

1.2 Problem Statement

As the number of cores increases in a single chip processor, several challenges arise: wire delays, contention for out-of-chip accesses, and core heterogeneity. In order to address these issues and satisfy the demands of the applications, future large-scale many-core processors are expected to be organized as a collection of Non-Uniform Memory Accesses (NUMA) clusters of heterogeneous cores. The work in this thesis is focused on many-core clustered architectures and more specifically to study: *(i)* how to achieve single application performance by exploiting the underlying architecture characteristics, *(ii)* resource isolation by forming performance domains which will guarantee applications performance and *(iii)* satisfy applications dynamic behavior on heterogeneous clustered many-core architecture. To address these issues but at the same time achieve high levels of applications performance in a clustered many-core architecture it is essential to answer the questions presented in the following sections.

1.2.1 Applications Parallelism

The increasing number of cores on a clustered many-core architecture can be exploited by applications with high degree of parallelism resulting in performance improvement. Porting an application for such architectures is not a trivial task but a joint task of considering both underlying architecture and application behavior. Memory-bound applications with high degree of parallelism can create an increasing number of memory requests, which must be satisfied without becoming a performance bottleneck. Decision Support System (DSS) workloads are known to be one of the most time-consuming database workloads that process large data sets. Traditionally, DSS queries have been accelerated using large-scale multiprocessors. Clustered many-core architectures provide specific benefits which could be exploited in order to achieve high levels of performance of such workloads. Different implementations of these workloads result on different performance, therefore architecture characteristics should be considered to achieve high levels of performance. In addition, enhancements of such architectures, *i.e.* on-chip memory shared among all cores, could be used as a prefetching buffer resulting on further improvement of the performance for such workloads.

Research Question 1: *How is it possible to achieve high levels of scalability and efficiency of memory demanding applications exploiting many-core architecture characteristics?*

1.2.2 Performance Domains

The use of virtualization has traditionally been as to divide the physical machine into different domains for three major purposes. First, to allow the installation of different Operating Systems (OS) on the same machine. Second, to provide isolation between users logged on to the different domains. Third, to improve the overall system utilization by offering more virtual processors than the ones available physically. The objective of virtualization was the better management of the complex underlying hardware as well as improving the utilization but not compromising on isolation. Nevertheless, these benefits usually come with a price - performance overhead. Obviously, the addition of an intermediate layer between the hardware and the application leads to some performance degradation, which should be minimized.

As the number of cores increases in multi-core processors, more applications execute at the same time. This may lead to interference between the different applications, and consequently a negative impact on their performance. The isolation properties provided by virtualization methods may offer performance predictability for the executed applications by eliminating in some cases the negative effects of co-execution interference. Therefore, introducing the benefits of applications isolation offered by virtualization techniques on HPC applications can result on performance guarantee and system utilization.

Research Question 2: *How is it possible to offer performance guarantees for the co-execution of multiple high-performance computing applications on many-core systems without adding significant overheads?*

1.2.3 Clustered Many-Core Architectures

The increasing number of processing units per chip results in a higher demand for "feeding" those units with both instructions and data. At the same time, neither the number of pins on the chip, nor the links to memory improve at the same rate as the number of cores. Moreover, the complexity of the interconnection network of large-scale multi-core architectures increases with the number of cores. The above mentioned multi-core issues result in limiting the scalability in terms of number of cores of these architectures. The proposed large scale clustered many-core architecture by Intel, also known as the Intel SCC [1] addresses the above limitations. Clustered many-core architectures consist of tiles of cores with private L1 and L2 cache, interconnected by a 2D-grid network. Off-chip memory requests are served by a number of memory controllers which are dedicated to a cluster of cores. In such architectures specific factors should be studied to define their impact to applications performance and consequently system throughput. More specifically, as the number of cores per cluster increases, so does their distance to the memory controller. This factor leads to non-uniform memory accesses (NUMA) and consequently influences the application execution time. Future clustered many-core architectures may consist of resources of different computational capabilities. This configuration can result on different performance and power efficient domains, which can satisfy different application requirements and therefore increasing system throughput and power efficiency. Given the application, determining appropriate resources for power-performance efficiency is not a trivial task. In addition, given the architecture characteristics of a clustered heterogeneous many-core architecture it is very difficult in practice to coordinate the scheduling operations during runtime. The key for determining the best matching core for a certain application is to find out the application's memory and computational requirements.

Research Question 3: *How can a runtime system dynamically chracterize and assign the best matching resources for multiple high-performance applications on a heterogeneous clustered many-core system?*

1.2.4 Modular Virtualization Layer

As technology advances and architectures change, tuning the same applications over and over for the new architectures becomes an overwhelming task. Also, by using the same core designs, manufacturers are able to produce many different processors, depending on the number of available cores and their configuration. A virtualization layer or hypervisor can hide the complexity and diversity of the hardware. This virtualization layer can operate as the manager of the underlying hardware hiding some complexity of the system from the Operating System. In other words, by offering this virtualization layer along with the hardware it is possible to offer a standard set of core services to the upper layers, such as thread scheduling, memory prefetching, and hardware reconfiguration. For example a regular Operating System could use the scheduling services provided by the virtualization layer to do the mapping of the tasks among the available cores. This mapping could be as simple as just randomly distributing the threads among the different cores or as complex as making architecture-aware decisions that, based on online monitoring information of the application behavior, are able to select the best matching cores available. The mentioned services are supported by a group of mechanisms transparent to the user and/or the Operating System.

1.3 Thesis Statement

In order for applications to exploit the performance benefits of multiple heterogeneous cores in a system you need a runtime environment that can help with different tasks such as data prefetching, performance isolation between co-execution, and best matching of resources determined dynamically. The vision is that future many-core chips will have a virtuallization layer that will support these and many more services transparently to the user.

1.4 Objectives and Contributions

1.4.1 Goal

The main goal of this thesis is to address the challenges raised for future many-core architectures. *First*, the architectural characteristics of clustered many-core architectures are exploited as to achieve high performance efficiency. *Secondly*, virtualization technique are explored as a way to achieve isolation and minimize the impact of interference on shared resources for co-executing applications. *Finally*, a scheduling policy is proposed for clustered many-core architectures having as an objective to satisfy the requirements of the applications in terms of available resources during runtime. To present the findings the proposed implementations are evaluated using applications from different domains. The objectives and contributions of this thesis are stated below.

1.4.2 Objective 1: Parallelism on a Clustered Many-Core Architecture

To answer *Research Question 1* (Section 1.2.1), the benefits of using future many-core architectures are exploited, more specifically on-chip clustered many-core architectures, on memory demanding applications. To achieve this goal the performance of the basic database algorithms parallelized are analyzed using the RCCE programming API [2] provided for the Intel SCC. Different implementations for the data parallel sequential scan, nested-loop and hash join query algorithms were implemented. Additionally, the impact on performance using a shared on-chip memory message passing buffer (MPB) as a prefetching buffer is also studied. The algorithms are the basis for the execution of standard representatives DSS queries taken from the TPC-H benchmark suite [3]. Real database workloads were selected, which represent different database algorithms, and their performance is evaluated on a real system, the Intel SCC experimental processor [4], using the proposed method of data prefetching.

Contribution: This work exploits the characteristics of clustered many-core architectures for the benefit of memory demanding applications. More specifically, the on-chip message passing buffer is used as a prefetching mechanism which stores data for all cores to be used when needed. Therefore, the off-chip memory accesses are minimized and eviction of useful data from L2 caches. The MPB is used in two ways: *(i)* as a whole, where each core writes only to its own MPB and reads from all and *(ii)* as 48 different buffers, where each core writes and reads only to and from its own MPB [5,6]. More details on the implementation and results of this work are presented in Chapter 2.

1.4.3 Objective 2: Guarantee Performance

To address *Research Question 2* (Section 1.2.2), virtualization is used as a way to split the multi-core processor into different *Performance Domains*. The objective is to provide a way for several applications to execute on the multi-core processor but at the same time to guarantee performance isolation. Therefore, an application executing on a certain performance domain will maintain its performance independently of the load in the rest of the multi-core processor. In addition, this solution is non-intrusive, *i.e.* it does not require any changes to the application and can thus be applied to any existing executable.

Contribution: The contribution of this work is firstly the analysis of the feasibility of such a solution and secondly to study the different virtualization methods, *i.e.* hosted and bare-metal virtualization. Performance penalty suffered by HPC applications when co-executing on top of different VMs is also studied [7]. More details on the implementation and results of this work are presented in Chapter 3.

1.4.4 Objective 3: Heterogeneous and NUMA-aware Scheduling

To address *Research Question 3* (Section 1.2.3), a runtime system is proposed for clustered many-core architectures which should determine the best matching core for a certain application by identifying the application's memory and computational requirements. It is important to acquire enough data to perform application classification and to classify the application as memory- or compute-bound so it is given the best matching core. In addition to the placement, the runtime should constantly monitor the behaviour of the applications. If changes are observed, for example resulting from an application entering a new phase, a classification phase is triggered in order to determine a new better placement. Since classification of applications is applied on a system with applications co-executing, this is achieved by indirectly observing how the performance is affected from the interference on shared resources.

Contribution: The main contributions of this work is: *(i)* propose a dynamic online classification methodology by determining the degree of memory- and compute-bound for each application, *(ii)* propose and implement a scalable dynamic scheduling policy for future heterogeneous many-core architectures, *(iii)* evaluate the scheduler using real applications applications (from SPEC benchmark suite) on a real many-core architecture (the 48-core Intel SCC processor) and *(iv)* study its scalability and extension over a simulated many-core architecture [8, 9]. More details on the implementation and results of this work are presented in Chapter 4.

1.4.5 Thesis Vision for the Future

Future multi-core processors will be composed of cores with different computational and memory capabilities, which are also able to change their configuration at runtime resulting in Morphable architectures. Morphable multi-core architectures are composed of several asymmetric cores which have the capability of changing their configuration at different levels for both their logic and memory elements. This results in a more efficient hardware platform that besides being able to adapt the application execution to the hardware underneath, is also able to adapt the hardware to the demands of the different applications and/or phases. Managing and exploiting such future large-scale systems, not only in terms of software but also the hardware architecture and design, is not a trivial task. The vision of this thesis for future many-core processors is that they will be coupled with a Virtualization Platform, which will be able to wrap the complexity of the underlying hardware and manage its resources transparently to achieve an improvement of the overall system efficiency [10].

Chapter 2

DSS Workload Parallelism on a Clustered Many-Core Architecture

The increasing number of cores on a clustered many-core architecture can be exploited by applications with high degree of parallelism resulting in performance improvements. Porting an application for such architectures is not a trivial task but a joint task of considering both underlying architecture and applications behavior. Memory-bound applications with high degree of parallelism can create an increasing number of memory requests, which must be satisfied without becoming a performance bottleneck. Decision Support System (DSS) workloads are known to be one of the most time-consuming database workloads that process large data sets [11]. Traditionally, DSS queries have been accelerated using large-scale multiprocessors. In this work the benefits of using future many-core architectures are exploited, more specifically on-chip clustered many-core architectures. To achieve this goal different representative data parallel versions of the original database scan and join algorithms are proposed. Additionally, the impact on the performance when on-chip memory, shared among all cores, is used as a prefetching buffer is also studied. For the experiments of this work the behaviour of three queries from the standard DSS benchmark TPC-H executing on the Intel Single chip Cloud Computer experimental processor (Intel SCC) is studied. Results show that parallelism can be well exploited by such architectures and how important it is to have a balance between computation and data intensity. Moreover, the experimental results show that performance improvement of *5x* and *10x* for the corresponding query implementation without data prefetching. Finally this work shows how the system could be used efficiently to achieve high power-performance efficiency when using the proposed prefetching buffer.

2.1 Motivation

The multi-core architecture is the *de-facto* standard in processor design. This architecture offers the benefit of an increased degree of parallelism to provide better performance, without the drawbacks of previous monolithic designs, such as high power consumption and complex design. As technology improves, the integration level increases leading to an increase in the number of cores per chip. While this results in a further increase of the degree of parallelism, it may not necessarily lead to improved performance, even when considering highly parallel applications. The increasing number of processing units per chip results in a higher demand for "feeding" those units with both instructions and data. At the same time, neither the number of pins on the chip, nor the links to memory improve at the same rate as the number of cores. Moreover, the complexity of the interconnection network of large-scale multi-core architectures increases with the number of cores. The above mentioned multi-core issues result in limiting the scalability in terms of number of cores of these architectures. The proposed large scale many-core architecture by Intel, also known as the Intel SCC [1] addresses the above limitations.

Database applications are of the most demanding workloads. More specifically, Decision Support Systems (DSS) database applications combine the processing of large data sets along with the computation of statistical information extracted from data. The goal is first to show the advantages that a future clustered many-core architecture, like the Intel SCC experimental processor [1], could have in a large scale data center that handles DSS applications. Secondly, the benefits of prefetching are presented for the studied workloads when a shared on-chip memory is used as a prefetching buffer. Finally the power-performance efficiency analysis of the system for the different query implementations is shown.

This work analyzes the performance of the basic database algorithms parallelized using the RCCE programming API [2] provided for the Intel SCC. Different implementations for the data parallel sequential scan, nested-loop and hash join query algorithms were selected. For the proposed implementations the impact on performance when a shared on-chip memory is used as a prefetching buffer is also studied. The algorithms are the basis for the execution of standard representatives DSS queries taken from the TPC-H benchmark suite [3]. Real database workloads, which represent different database algorithms, were selected and their performance is evaluated on a real system, the Intel SCC experimental processor, using the proposed method of data prefetching. Results show performance improvement by factors of *5x* to *10x* when data prefetching is used. Moreover, with a small performance loss high benefits in power consumption are gained resulting to high power-performance efficiency.

2.2 Related Work

Different works study the performance evaluation and optimization of database workloads. Porting and evaluating the performance of such workloads in different systems is being of a large interest due to the fact that large data centers are executing such workloads and performance and power efficiency is of their most interest. Data prefetching is a technique widely used for reducing the memory latencies by fetching data to the processors cache before it is requested in order to avoid misses that would otherwise occur. Data prefetching can be done automatically in hardware [12], where the prefetcher predicts the next data to be requested, or in software where data requests are explicitly placed by the programmer of the compiler in the code [13]. An example of data prefetching for accelerating the execution of database workloads is in [14]. Pre-execution [15] is an alternative prefetching mechanism using an extra thread called helper thread that executes portions of the code ahead of the execution threads. Trancoso *et al.* [11] investigated the acceleration of decision support queries when executed on Cell/Be [16] and GPUs using Rapidmind [17] as a common platform. For their evaluation they used different workloads from the TPC-H benchmark suite. They implemented a nested loop join and a hash join algorithm exploiting the streaming processing model in order to optimize the performance of the queries. Their experimental results show speedups up to 21x. In [18] the authors presented an efficient way of memory copy operations on the Intel SCC experimental processor where they manage to use the LUT entries of each core in order to direct forward the necessary data to the corresponding core. In [19] the authors investigate the performance behaviour of a pipelined parallel sorting ported for the Intel SCC experimental processor showing that a combination of pipelined mergesort and sample sort best fits in such architectures.

Different works have focused on dynamic voltage and frequency scaling [20]. Isci *et al.* in [21] show multiple power domains offer power savings in CMP systems over a single power domain. Ioannou *et al.* in [22], propose a hierarchical power management scheme applicable to reduce the energy consumption on many-core architectures by reducing the running frequency/voltage of the cores. The experiments conducted by the authors on an Intel SCC machine show that their power management scheme is capable of an average reduction of energy consumption by 11.4% while the execution time is increased on average by 7.7% compared to the baseline execution without the use of the power management scheme. In [23] the authors presented an analysis of power and energy consumption of Intel SCC research processor. The authors used time and power profiling of a very simple parallel application. They also study the effects of power and energy consumption of the system by varying the number of cores, clock frequency and voltage level.

This work studies the performance and parallelism scalability of database queries algorithms using different techniques in order to optimize their performance like data prefetching. Moreover, power-performance efficiency analysis of database algorithms using different implementations is presented. Finally, experiments were executed on a representative many-core architecture platform, the Intel SCC, and they show how the selected workloads can be benefit in such architectures.

2.3 Intel SCC Clustered Many-Core Architecture

The Intel SCC experimental processor is a 48-core 'concept vehicle' created by Intel Labs as a platform for many-core software research [1]. This processor consists of 24 dual-core tiles interconnected by a 2D-grid network. The tiles are organized in a 6x4 mesh with each tile containing:

- Two P54C cores with 16KB L1 and 256KB L2 cache dedicated to each core;
- A 16KB Message Passing Buffer (MPB) for storing messages to be sent to other cores (8KB per core);
- A Traffic Generator for testing the performance capabilities of the mesh network;
- A Mesh Interface Unit (MIU) to connect to the network;

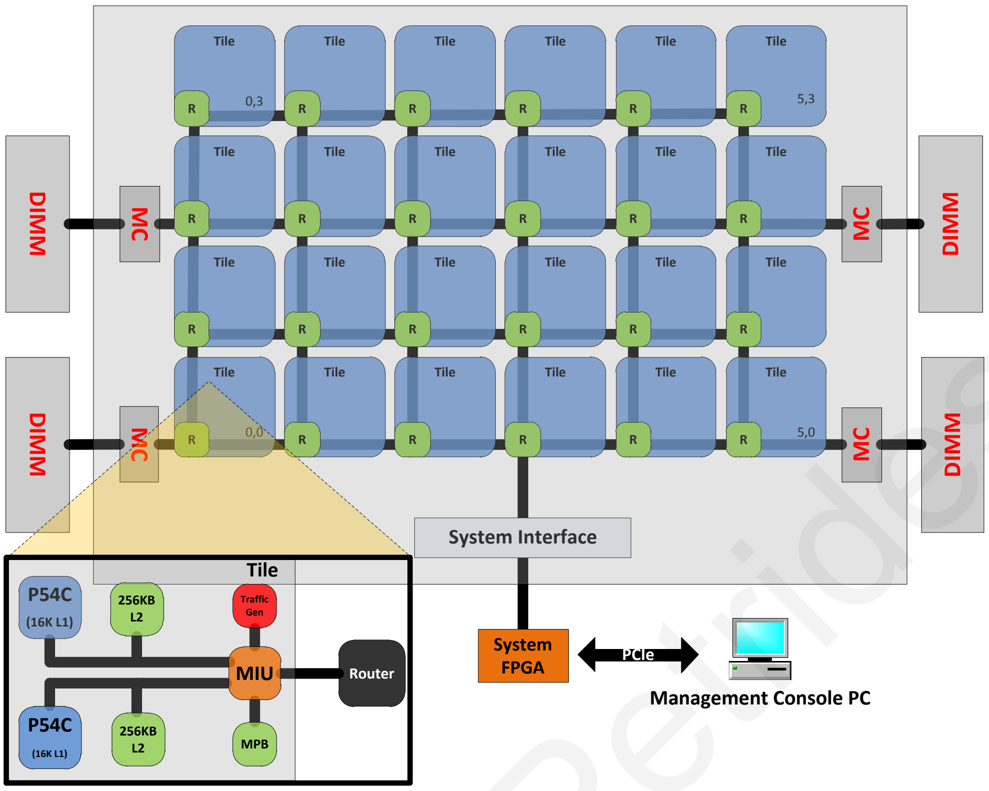


Figure 2.1: Intel SCC Architecture

The MIU connects each tile to a router, to finally create a mesh interconnection network. The MIU is responsible for packetizing data that will go onto the network and de-packetizing data that come from the network. This unit is shared by the two cores in a round-robin scheme [24].

The maximum main memory the system can support is 64GB. The 32-bit memory addresses of the core are translated into system addresses by the MIU through a lookup table (LUT) [24]. The systems' main memory is located outside the chip and four DDR3 Memory Controllers (MC) are used to move data on and off chip. Figure 2.2 presents an overview of the SCC Memory Architecture and how the programmer can view the system and the core's memory through the RCCE message passing API [24]. The SCC supports both distributed and shared memory programming models and the systems' memory is configured and separated in four regions:

- Private off-chip memory: Each core's LUT is configured such that a specific region of the off-chip DRAM (equally divided to all) is only accessible by that core.

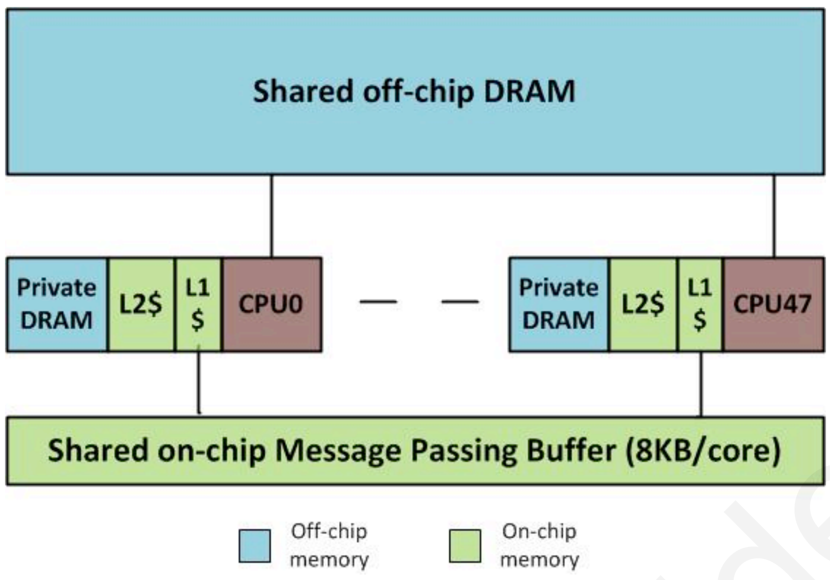


Figure 2.2: Intel SCC Memory Architecture as used by the programmer through the RCCE message passing API

- Shared off-chip memory: This region of the off-chip DRAM is mapped by all LUTs and all cores have direct access to it through any MC. These data are not cached as the system does not provide cache-coherence;
- Shared on-chip memory: Also called Message Passing Buffer (MPB), this on-chip SRAM is cached in the L1 caches of the cores;
- L2 cache: used only by the private off-chip memory

2.3.1 On-Chip Shared Memory

As described above the shared on-chip memory (MPB) is used for messages exchanged between cores where each core has a chunk of 8KB adding in a total 384KB of shared on-chip memory. Data from MPB are cached in cores' L1 cache. Except from a small amount of the MPB that is used by the system, it is possible to use the rest of it to move data closer to the cores and therefore minimize the delays of accessing data previously located to the off-chip memory.

In this work is studied how this shared on-chip memory can be used as a prefetching buffer. More specifically, it studies how data for all cores can be stored in this buffer in order to be used when needed. Therefore minimizing the off-chip memory accesses and eviction of useful data from L2 caches. MPB size is limited to 384KB and each core writes to its own chunk of 8KB. Therefore the MPB is used in two ways: (i) as a whole, where each core writes only to its' own MPB and reads from all and (ii) as 48 different buffers, where each core writes and reads only to and from its' own MPB. Section 2.5 describes in detail how the MPB is used as a prefetching buffer for the different query algorithms.

2.4 Database Workloads

This work is focused on the execution of the basic database algorithms and their parallel implementation. As such, the queries analyzed in this work were implemented as programs that execute the operations determined by the queries and their results were validated. Three different queries from TPC-H benchmark suite [3] of different complexity and demands have been ported. More specifically Queries 3, 6 and 12, which from now on are referenced as Q3, Q6 and Q12. Figures 2.3, 2.4 and 2.5 describe the three queries in the original SQL code. The selected queries are of different complexity both in terms of the amount of data uses and in terms of the operations performed on the data [25]. More specifically, the selected queries include representative operations performed in DSS workloads. In particular, simple table scan performing an aggregate sum on specific field is included in *Q6*, whereas *Q12* joins two tables and performs a count on the items satisfying the query condition. Finally, *Q3* joins three tables and selects a specific field when the query conditions are satisfied. Therefore, the selected query algorithms that encapsulate the representative operations of DSS workloads both in terms of operations and in terms of combining different data sets for obtaining the desired results. The processing data is formatted in two ways in order to evaluate different implementations of the query algorithms.

In the first format data are stored row-wise, *i.e.* all attributes of a particular record are stored in the same *row* of a two-dimensional array. Let's consider a table (Table A) which is composed of records containing three attributes: attr1, attr2, and attr3. For each record a new row is created that stores all its attributes.

Another format of data is hashing the data according to the key, primary or foreign, on which tables can be joined. More specifically, discrete linked lists of records are created based on the tables' key on which the join operation is performed.

For example, consider Figure 2.6 (c) and (d), Tables' A records are hashed according to their primary key whereas Tables' B records are hashed according to

```
select
        sum(l_extendedprice*l_discount)
        as revenue
from
        lineitem
where
        l_shipdate >= date '[DATE]'
        and l_shipdate < date '[DATE]'
        + interval '1' year
        and l_orderkey = o_rderkey
        and l_discount between
        [DISCOUNT] - 0.01 and + 0.01
        and l_quantity < [QUANTITY];
```

Figure 2.3: TPC-H Q6. The parameters used were: DATE=2005, DISCOUNT=10, and QUANTITY=1000000.

```
select
        sum(case when
                o_orderpriority = '1-URGENT' or
                o_orderpriority = '2-HIGH'
                then 1 else 0 end)
         as high_line_count
from
        orders, lineitem
where
        o_rderkey = l_orderkey
        and l_shipmode in
        ('[SHIPMODE1]', '[SHIPMODE1]')
        and l_commitdate < l_receiptdate
        and l_shipdate < l_commitdate
        and l_receiptdate > date '[DATE]'
        and l_receiptdate > date '[DATE]'
        + interval '1' year;
```

Figure 2.4: Simplified version of TPC-H Q12. The parameters used were: SHIPMODE1=1, SHIPMODE2=2 and DATE=2009.

their foreign key. The join operation between Table A and B can then be performed directly.

```
select
        l_orderkey ,
from
        customer , orders , lineitem
where
        c_mktsegment = '[SEGMENT]'
        and c_custkey = o_custkey
        and l_orderkey = o_rderkey
        and o_orderdate < date '[DATE]'
        and l_shipdate > date '[DATE]';
```

Figure 2.5: Simplified version of TPC-H Q3. The parameters used were: DATE=2007 and SEGMENT=3.

2.5 Algorithms Implementations

The query algorithms implemented for the purpose of this work are: *(i)* data-parallel sequential scan, *(ii)* nested-loop join and *(iii)* hash join. More details on their implementation follows.

2.5.1 Data-Parallel Sequential Scan (DPSS)

Given the data layout as presented above, for this work, the simple sequential scan algorithm is used as to exploit both load balancing and locality while traversing the data. For the purpose of this work, all records are traversed and the records' attributes are checked against a certain condition. The condition may be a simple attribute comparison or a complex boolean function. This operation is mapped to the Intel SCC by implementing the condition to be tested and by sending to each core the input parameters which are the data streams that are used to evaluate the scan condition.

2.5.2 Parallel Nested-Loop Join

For the join operation the nested-loop join algorithm is used. Each table has a number of rows equal to the number of records and the data of each record are organized in columns. To perform this operation each record of the outer loop is compared with all the records of the inner loop in an iterative way. Parallelization is achieved by splitting the data of the outer loop to all cores.

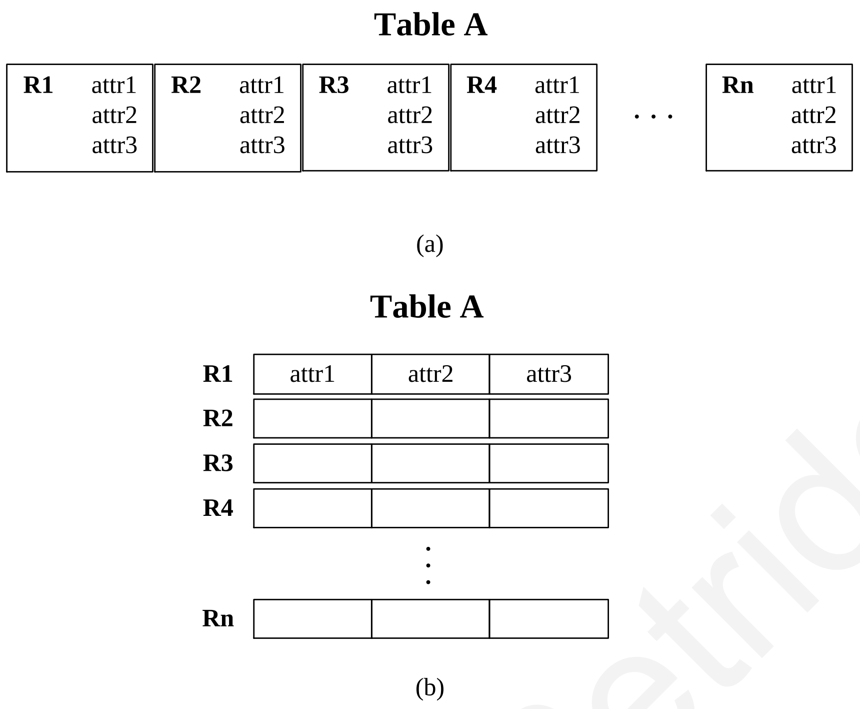


Figure 2.6: Table A: (a) logical and (b) physical data organization for parallel nested-loop join operation.

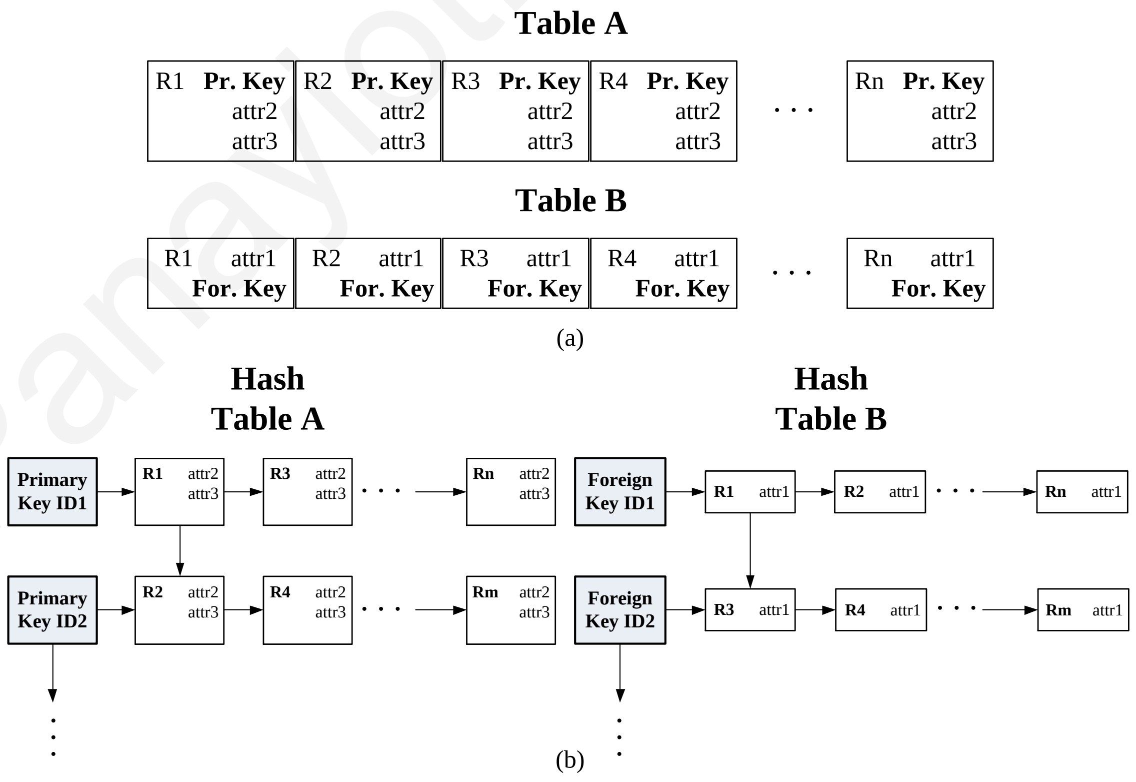


Figure 2.7: Table A: (a) logical and (b) physical data organization for hash join operation.

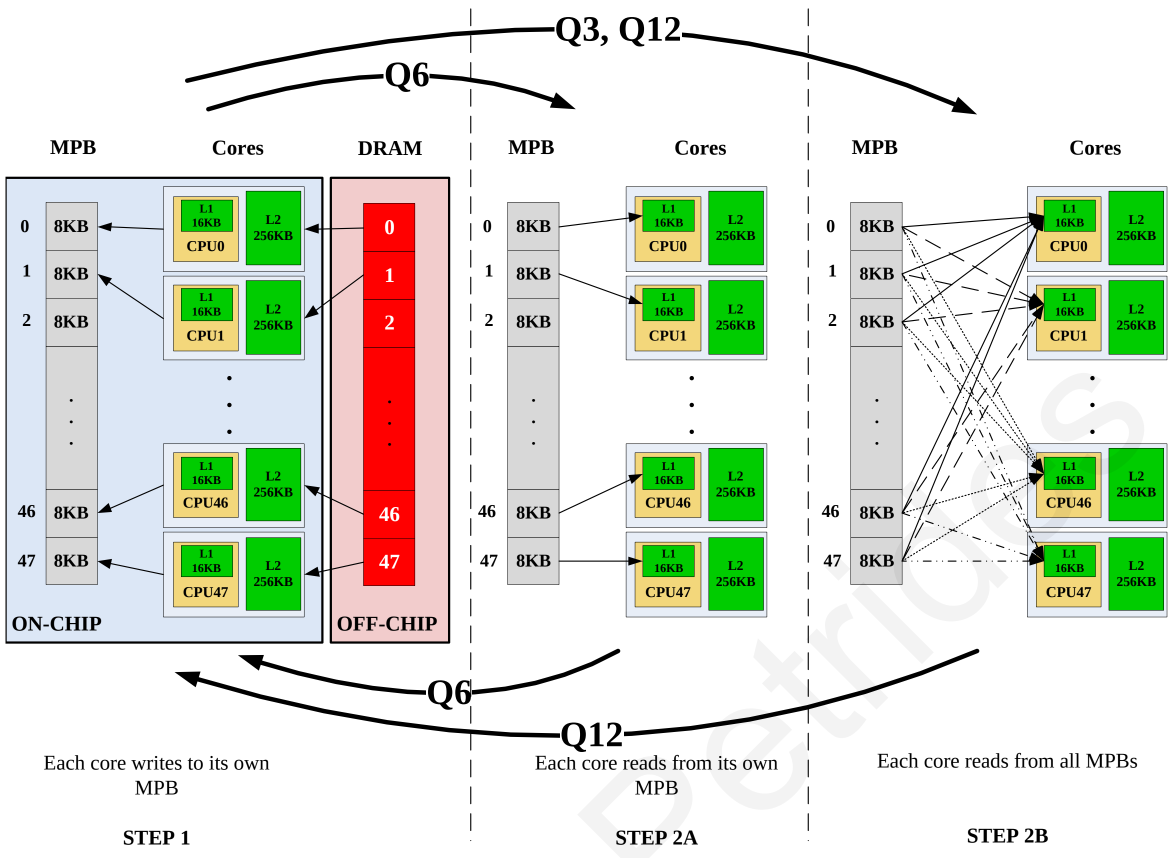


Figure 2.8: Description of data prefetching using the MPB.

2.5.3 Hash Join

Another implementation of the join operation that is evaluated for the selected queries algorithms is the hash-join algorithm. In this case all records of each table are organized according to the key that the join operation is performed on as described in Section 2.4. Therefore the join operation is performed on each lists' key and if the condition is satisfied then all its records are examined. In the case of Q12 where only two tables are joined this operation is performed only once and if the condition is satisfied then it proceeds with the operations defined in the where clause of the query. In the case of Q3 although there are three joined tables. First Tables A and B are joined, as in Q12, and the results produced from this operation are forwarded to the second step where these results are joined with Table C using the same method of hashed data. Parallelization of the hash join implementation of the queries algorithms is achieved by splitting the data in the first level of join. Meaning each core takes a chunk of data of the first table that is joined in the algorithm. For example in Q12 where Tables A and B are joined, Table A is split among cores. In the case of Q3 where Tables A, B and C are joined Table A is split among cores.

2.5.4 Data Prefetching

In order to use data prefetching and minimize the number of off-chip memory accesses the MPB of the system is used. To achieve this the previously mentioned algorithms in Sections 2.5.1, 2.5.2 and 2.5.3 are implemented using the MPB as a prefetching and storing buffer for most commonly used data, in an effort to optimize the performance. Data fetched or read from the MPB are controlled at user level by specifying which data and when are to be used.

For the Q6 query which is a simple scan operation, the MPB of each core is used to fetch and store portions of data, 8KB long, from main memory. From there on these data are copied directly to L1 cache and the necessary operations are performed. In this case each core writes and reads data only to and from it's own MPB. In Figure 2.8 the representation of this algorithm is depicted. In Step 1 each core copies data from the off-chip main memory to its' local MPB and in Step 2A it copies those data to its' L1 cache for calculation. These two steps are iterative steps until all data are processed.

For the Q12 query the data prefetching scheme is used only for the nested-loop join implementation. The reason that the prefetching could not be used on the hash-join implementation is related to the organization of the data and that moved data are moved from one table to the MPB then the hashing implementation would no longer exist. On the other hand if the hash table is moved into the MPB there would still be a need to go off-chip in order to access the data directed by the hash table (currently located in the main memory). This would result in a larger overhead, minimal cache misses reduction and eviction of necessary data from L2 cache, since main memory is cached in L2. For the nested-loop join algorithm of Q12 the MPB is used to store data from the inner-most table (Table B) and let the outer-most table (Table A) to be cached in L2. In Figure 2.8, Q12 algorithm for data prefetching uses Steps 1 and 2B. Each core will copy data from Table B to its' local MPB (Step 1) and then each core will read and perform calculations on data from all the MPB chunks (Step 2B). Step 1 and 2B are iterative because Table B cannot fit in the total of 384KB of the systems' MPB, so as soon as all cores finish Step 2B they will repeat this process again until all data are processed.

For the Q3 query the data prefetching is used on both nested-loop join and a hybrid implementation of hash-join and nested-loop join. For the nested-loop algorithm the inner-most table (Table C) is stored in the MPB. Table C is chosen to be prefetched and stored in the MPB since it is the table that is most frequently used and thus resulting in a significant reduction of the last level cache misses. Also, it has the smallest size and in this case it could fit for both input sizes in the MPB of the system. For the hybrid implementation, where both hash-join and nested-loop join are used, the hash-join operation is performed for the first join operation (Tables A and B) and for the records satisfying the query condition a nested-loop join with Table C is performed. In Figure 2.8 Steps 1 and 2B depict this implementation showing that Step 1 in Q3 will only be executed once and each core will copy a portion of Table C to its' local MPB. In the case that Table C does not fit in the MPB the algorithm will become iterative and continue fetching data from main memory just as it happens with Q12. Finally in Step 2B each core will read and perform calculations on data from all the MPBs of the system.

2.6 Experimental Setup

For the evaluation of this work the Intel SCC experimental processor is used, RockyLake version [1]. The operating system used for the Intel SCC cores is the default Linux kernel provided by the RCCE SCC Kit 1.4.0 [2]. The host PC, responsible for controlling the applications execution on the Intel SCC processor, is configured with Intel Core i7 processor 3.7GHz and 4GB of main memory. For porting and executing the applications on the SCC the RCCE 1.4.0 toolchain was used.

The workloads used for this work were selected from the TPC-H queries as described in Section 2.5. Different input sizes were used for evaluation in order to study their performance scalability. The input data sets were generated using the *dbgen* tool. The input sizes 0.01 and 01 as well as the number of tables used for each query execution are: *(i)* Q3 use of 3 Tables of total size of 4.24MB and 93.56MB, *(ii)* Q6 use 1 Table of size 4.24MB and 93.56MB and *(iii)* Q12 use of 2 Tables of total size 3.74MB and 91.14MB.

For the purpose of this work the power consumption is also measured of the system in order to calculate the power-performance efficiency of the system. In terms of calculating the power consumption of the chip an application is developed

Table 2.1: *TPC-H Queries input sizes.*

Query	Tables	Input Size 0.01	Input Size 0.1
Q3	3	4.24MB	93.56MB
Q6	1	3.71MB	74.24MB
Q12	2	3.74MB	91.14MB

that measures the power consumption using the same technique used by the SCC GUI performance meter and involves reading the FPGA emulated registers that hold the appropriate values. The measurement is calculated using the product of the total SCC chip voltage and the total current flowing through the chip. To enhance the precision of the used meter the power measurement is being collected several times every second during the whole execution time of each scenario. At the end of the scenario execution the average power consumption was calculated from the measurements done during the execution. For the experiments of this work the frequency of the cores is scaled to three different frequencies: (i) 100MHz, (ii) 266MHz and (iii) 533MHz. The frequency scaling is performed before the execution of each scenario using the library provided by the RCCE toolchain. In order to calculate the power-performance efficiency of the system the following metric is used:

$$NormalizedEfficiency = \frac{ExecutionTime*Power_{baseline}}{ExecutionTime_i*Power_i} \tag{2.1}$$

The experimental results were normalized using the formula in 2.1 for the different implementations to the single core execution of the base line scenario of each query executing at 533MHz (where *i* are the different implementations of the query algorithms, *i.e.* Nested-Loop Join, Hash Join etc). Higher the results the better.

2.7 Experimental Results

2.7.1 Performance Evaluation

For the first analysis the performance behaviour is compared of the different queries algorithms executed on the Intel SCC experimental processor as described in Section 2.5 and their scalability is studied for the target architecture. Moreover, it is

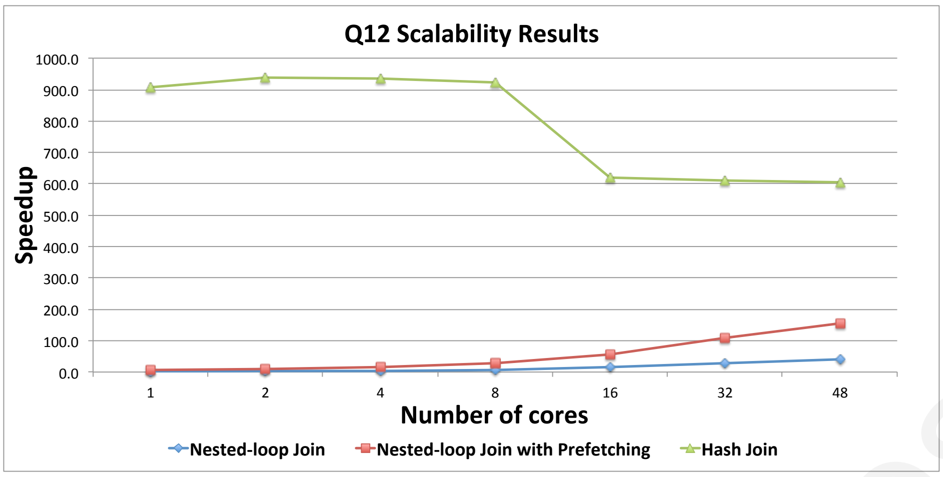


Figure 2.9: Normalized scalability of Q12 query.

studied how the data prefetching scheme proposed can improve their performance. The results presented herein are for 533MHz cores' frequency due to the limited space available, even though 266MHz and 100MHz show the same behavior.

In Figure 2.9 the performance behavior and scalability of the different implementations of Q12 query is presented. In Figure 2.9 execution times are normalized to the corresponding execution of the nested-loop join implementation for a single core. Results show first of all the hash join algorithm outperforms the nested-loop join implementation by a factor more than *10x*. This performance difference can be explained from the way data are mapped and the efficiency of the hash join implementation. In the case of the loop-join implementation each record of the outer table must be compared with all the records of the inner table, therefore resulting to more comparisons in contrast to the hash join implementation. The second important observation is when the prefetching scheme is used for the nested-loop join algorithm. It can be observed that up to 5x performance improvement can be gained compared to the nested-loop implementation as a result of devoting MPB for data that are mostly used. In particular data are prefetched to MPB and these data are stored without being influenced from evictions of L2 cache, therefore data reuse can be achieved without any additional main memory accesses. Using performance counters and monitoring the L2 cache of the core it was observed that the data prefetching implementation reduces the L2 cache misses by a factor of *700*. Even though using data prefetching, the performance of the hash join implementation could not be outperformed as a result of their significant difference to their data organization and algorithm implementation. As for the scalability of the different implementations,

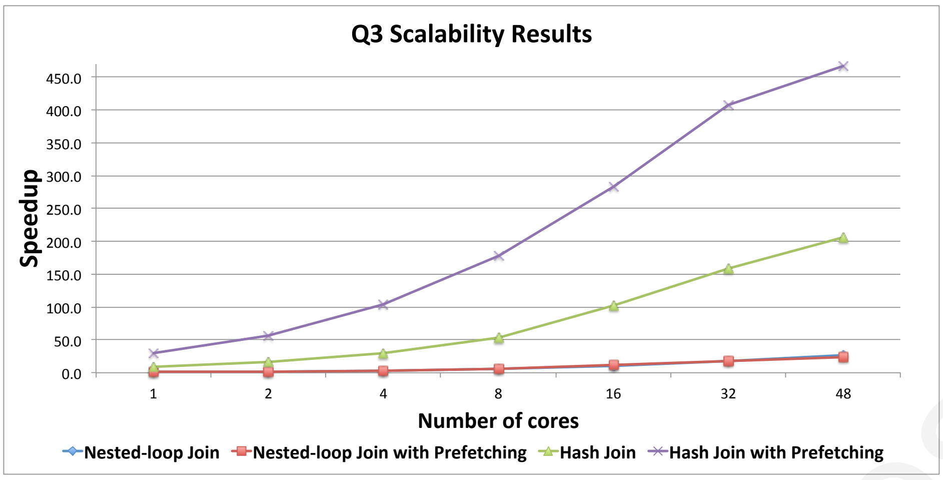


Figure 2.10: Normalized scalability of Q3 query.

it is possible to see that nested-loop join (with or without data prefetching) scales well whereas hash join implementation is stable to the number of cores. This can be explained from the fact that the hash join implementation algorithm complexity is limited and data transfers dominate the computation time, in a ratio of 1:20 computations over data transfers, resulting in no performance improvement as the number of cores increases.

In Figure 2.10 the performance behaviour and scalability of the different implementations of Q3 query is presented. In Figure 2.10 execution times are normalized to the corresponding nested-loop join implementation on a single core. The first observation is that the nested-loop join implementation using data prefetching does not improve the performance compared to the implementation without data prefetching. This is explained from the fact that the table stored in the MPB can fit in the L2 cache of the core compared to the limited size of the MPB. This results in higher overheads on fetching data from the main memory to the MPB until all data are processed. The difference to the execution times when using data prefetching is due to two main factors. First in the case of scenarios with 4,8 and 48 cores, records are equally divided among cores, as explained previously, but the number of each cores records that satisfy the condition varies among cores in an average of 20%. In addition, there is the overhead of the prefetching scheme. More specifically, data chunks of 8KB are always read from MPB regardless if they satisfy or not the where clause condition. Instead in the nested loop-join implementation data from main memory will only be read until the where condition is satisfied. In the case where the condition is not satisfied or all data must be read the two implementations converge.

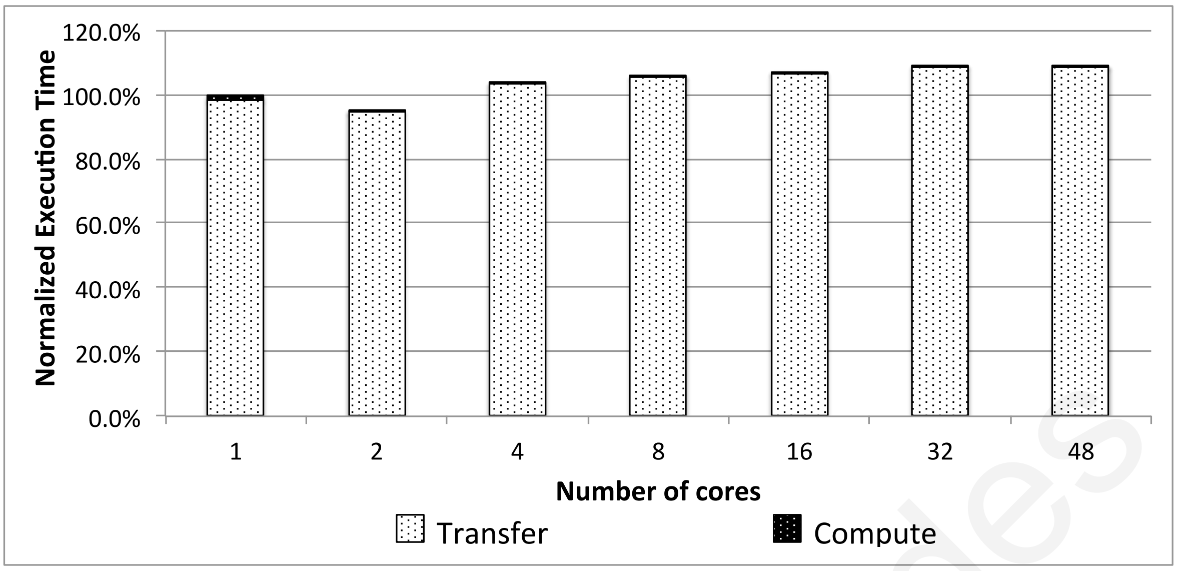


Figure 2.11: Data-parallel sequential scan (Q6) normalized execution time and breakdown.

Another important observation is the fact that the hash join implementation outperforms the nested-loop implementation having the behaviour of Q12. The most important observation is the fact that the hybrid implementation of Q3 using first hash join and the data that satisfy the condition are then passed to a nested-loop join where the table on which they are compared are already stored in the MPB. This implementation shows important performance improvement over the other implementations as a result of gaining the most from both the hash join implementation but also from the prefetching scheme.

It is also important to observe the scalability of the different implementations of the Q3 query. A good scalability can be observed, but not at the same level as the scalability of Q12 query. This could be explained by comparing the two queries algorithm complexity over the data sizes processed. More specifically Q12 is a well balanced query between computation and data size and therefore shows a better scalability compared to Q3 which has near the same data size but higher complexity than Q12. More specifically the ratio of computation over data transfers of Q12 is *5x* larger than the one of Q3.

In Figure 2.11 the normalized execution time of the data-parallel sequential scan implementation for Q6 is presented. The execution time is normalized to the execution time of the corresponding sequential scan on a single core. It can be observed that no significant performance improvement can be achieved with the use of data prefetching. This is a result of the simplicity of the query algorithm where data accesses dominate the computation time with a ratio of 1:30 of computations over data

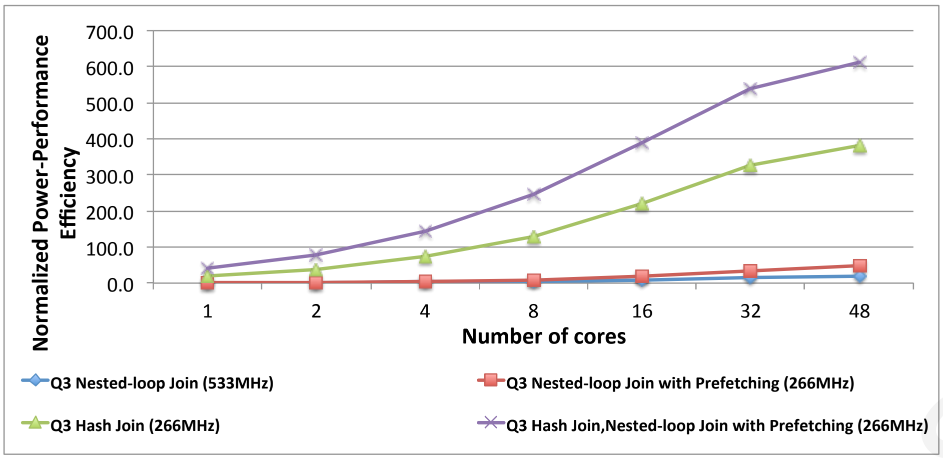


Figure 2.12: Normalized power-performance efficiency of Q3 query.

transfers. Also the fact that no data reuse is made in this algorithm does not allow the prefetching scheme to improve its performance opposed to the data prefetching implementations of Q3 and Q12 queries.

2.7.2 Power and Performance Efficiency

Another aspect of this work is to study the power-performance efficiency of the executed query implementations on the Intel SCC research processor. For this set of experiments the frequency of cores is scaled to 266MHz and 100MHz. As defined in equation 2.1 the higher the results the better for the different scenarios investigated. The most relevant results are depicted.

In Figure 2.12 the power-performance efficiency results of Q3 implementations is presented using as a baseline the nested-loop join implementation without data prefetching. Results show that high power-performance efficiency can be achieved if the proposed hybrid implementation is used, which is consist of the hash join and nested-loop join using prefetching. The results of the proposed hybrid implementation of Q3 query show that even if the frequency of cores is scaled to 266MHz high power-performance efficiency can be achieved compared to the other implementations of the same query. More specifically it can be observed that *600* more power-performance efficiency improvement can be achieved with the hybdrid implementation using data prefetching, where even though there is a small loss in performance high benefits in power consumption are gained.

In Figure 2.13 the power-performance efficiency results of Q12 implementations is presented using as a baseline the nested-loop join implementation without data

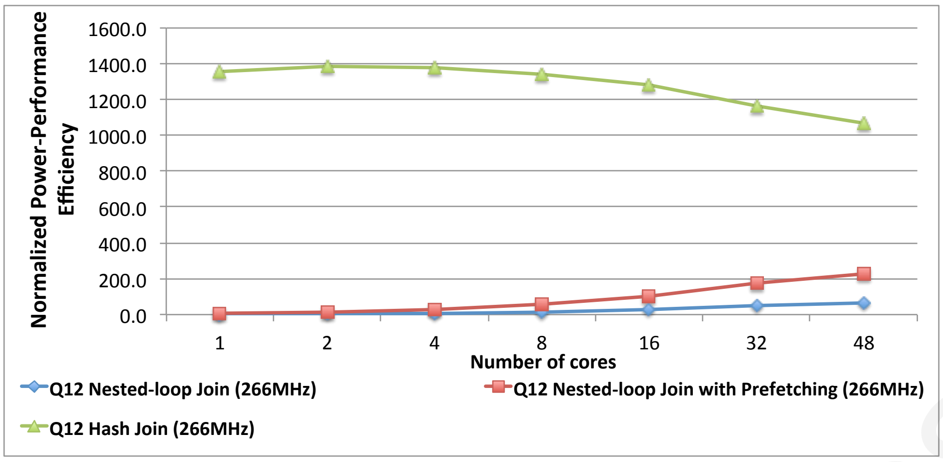


Figure 2.13: Normalized power-performance efficiency of Q12 query.

prefetching. Results depict that high power-performance efficiency, increased by a factor of up to *1400*, can be achieved using the hash join implementation of Q12 even if cores' frequency is scaled to 266MHz. Results show that up to 4 cores the power-performance efficiency of the selected implementation remains stable and afterwards reduces. Therefore higher system execution efficiency can be achieved if executing multiple instances of the same implementation in fewer cores per instance. Another important observation that can be extracted from these results is the high efficiency of the nested-loop join implementation using data prefetching over the nested-loop join implementation without the use of prefetching.

In Figure 2.14 the normalized power-performance efficiency of Q6 implementations using as a baseline the data-parallel sequential scan with no data prefetching is presented. Results show that up to 4 cores the power-performance efficiency of the selected implementation remains stable and afterwards reduces. It is relevant to mention that the two implementations do not differ much between then either using data prefetching or not. It is also important to mention that the system is more efficient and it can result in higher throughput if a single core scaled to the 266MHz is used for multiple execution instances of the same algorithm.

It is important to mention that if the target is high power-performance efficiency, the Intel SCC experimental processor cores frequency can be scaled down. Also different implementations are more suitable at each case depending on the complexity of the algorithm and their data sizes.

Overall the results were very encouraging for the use of data prefetching for future large-scale many-core processors even for demanding database applications.

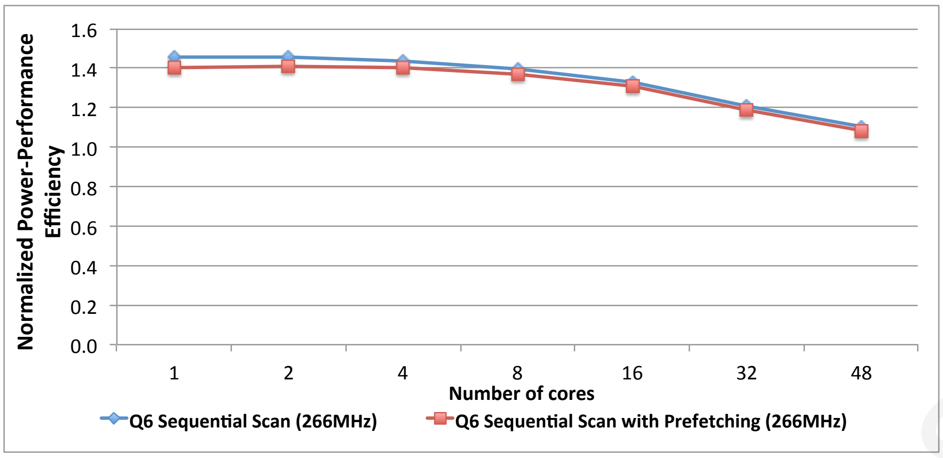


Figure 2.14: Normalized power-performance efficiency of Q6 query.

Performance improvement by factors of *5x* to *10x* when data prefetching is used are observed. Moreover, the scalability of such workloads is studied in a representative many-core architecture, the Intel SCC experimental processor, and this work show that the query algorithms that benefit the most from such architectures are the ones which balance well the ratio of computation over data transfers. Finally, this work analysis shows that with a small performance loss high benefits in power consumption can be gained, resulting to high power-performance efficiency.

2.8 Summary

Database applications are of the most demanding workloads. Three different queries were ported from the TPC-H benchmark suite on the Intel SCC experimental processor and their performance behaviour was studied when data prefetching is applied using the on-chip shared memory of the system. Experiments depicted that when there is no data reusage on the query algorithms (Q6) data prefetching shows no significant improvement. For medium complexity query algorithm with high input data size (Q12) nested-loop join algorithm using data prefetching can achieve up to 5x speedup. Although in this case hash join implementation is more efficient due to the simplicity of its algorithm. Finally for high complex queries in terms of the operations performed and high input data size (Q3) using a hybrid implementation of hash join and nested-loop join with data prefetching it is possible to improve performance by a factor of *10*.

Another aspect of this work was to evaluate the power-performance efficiency of the different queries implementations in order to investigate the most efficient implementation. Results showed that in the case of simple query algorithms, like Q6 and Q12 hash join implementation, scaling down the systems' cores frequency and reducing the number of cores (executing the respective implementation) can achieve both high power-performance efficiency and throughput when executing the query in multiple instances on the system. Finally, in the case of Q3 the results show that high power-performance efficiency can be achieved for the hybrid hash join and nested-loop join using data prefetching implementation even if cores frequencies are scaled to 266MHz compared to the other implementations.

Chapter 3

Multi-Core Performance Domains for HPC Applications

As the number of cores increases in multi-core processors, some applications benefit from the increasing number of parallel cores, in many cases the use of many-core processors will be for the co-execution of multiple applications. This might happen because of the limited degree of parallelism of the applications or in order to achieve higher throughput and resource utilization. In this chapter a simple and non-intrusive approach is presented which guarantees performance isolation for High Performance Applications. This is achieved using virtualization by creating multiple virtual machines on the same processor, which can be seen as different Performance Domains. While previously this technique has been explored for increasing utilization, in this work it is exploited for improving performance of multiple co-executing applications. For the purpose of this work two different virtualization approaches are studied: (i) conventional hosted virtualization and (ii) bare-metal virtualization. To study the feasibility of this technique, the performance of applications when executing within a virtual machine is analyzed. The isolation properties provided by both virtualization methods offer performance predictability for the executed applications. Experimental results show that the performance overhead of executing on a virtualized environment is not significant, with the bare-metal virtualization resulting in an overhead of only 3%. Most importantly, virtualization is able to eliminate in some cases the negative effects of co-execution interference, thus applications running on virtual machines may achieve a better performance than running natively on the system.

3.1 Motivation

The use of virtualization has traditionally been as to divide the physical machine into different domains for three major purposes. First, to allow the installation of different Operating Systems (OS) on the same machine [26]. Second, to provide isolation between users logged on to the different domains. Third, to improve the overall system utilization by offering more virtual processors than the ones available physically [27]. This type of virtualization was used extensively in the times of the mainframe computers. The objective of virtualization was the better management of the complex underlying hardware as well as improving the utilization but not compromising on isolation. Nevertheless, these benefits came with a price - performance penalty. Obviously, the addition of an intermediate layer between the hardware and the application leads to some performance degradation.

Given the performance issue and the fact that after the mainframes, most systems were personal computer (PC) based, virtualization was not very popular for those smaller systems. Nevertheless, recent technology advances are leading again to the use of virtualization. First there are large data-center systems that are designed for peak use and therefore suffer from very small to average utilization. While these systems store critical data for the business, the owners would like to open them to external users as to increase the utilization. One such example is Amazon and their EC2 system [28]. The performance penalty of virtualization, in this case, is not that relevant compared to the benefits obtained such as the users isolation and improved system utilization.

At another level, important changes in the processor technology and architecture are observed. In order to overcome the power and design complexity limitations, the industry has shifted the processor architecture from the traditional monolithic single core to the multi-core. As the number of available cores increases in the processor, there are opportunities to execute applications with larger degree of parallelism, or alternatively executing more applications at the same time. A naïve approach using existing OS may lead to interference in the performance of the different applications that are running simultaneously. This becomes a serious issue if someone is interested to use part of the multi-core resources for the execution of HPC applications. On the other hand, as the number of cores increases, the processor becomes harder to manage. Tasks such as scheduling, fault-tolerance and thermal violation avoidance become more complex.

In this work the use of virtualization is proposed as a way to address the above mentioned issues. In particular it is proposed to use Virtual Machines (VM) as a way to split the multi-core processor into different *Performance Domains*. The objective is to provide a way for several applications to execute on the multi-core processor but at the same time to guarantee performance isolation. Therefore an application executing on a certain performance domain will maintain its performance independently of the load in the rest of the multi-core processor. In addition, this solution is non-intrusive, *i.e.* it does not require any changes to the application and can thus be applied to any existing executable. The proposed solution of using a VM as to implement a *Performance Domain* is trivial. Nevertheless, the contribution of this work is firstly the analysis of the feasibility of such a solution and secondly to study the different virtualization methods, *i.e.* hosted and bare-metal virtualization. More specifically, this work is focused on analyzing the performance penalty suffered by HPC applications when co-executing on top of different VMs.

For the experiments of this work applications from the Princeton Application Repository for Shared-Memory Computers suite (PARSEC) [29] are used. These are multithreaded programs focusing on emerging workloads and designed to be representative of next-generation shared-memory programs for chip-multiprocessors. They were executed on top of Oracle VirtualBox [30] to study hosted virtualization and Xen hypervisor [31] in order to study bare-metal virtualization, on a system with two quad-core processors. The results observed show that the different characteristics of each application have a considerable impact on the penalty suffered by the execution on a virtualized machine (VM) and varies according to the virtualization method used. More specifically, this performance penalty ranges between 10 and 37%, on hosted virtualization, and 1 to 3 %, on bare-metal.

Finally, the creation of different domains using separate virtual machines on the same system show that these virtual machines are able to satisfy the isolation properties even in terms of performance for HPC applications. Moreover the interference of application co-execution on a native system is limited due to the virtualization properties provided. Therefore, in several cases the execution on a virtual system achieves better performance than on the native system. All these results lead to conclude that it is possible to use virtualization as a way to reserve a portion of the multi-core resources for HPC execution independently of the load on the rest of the multi-core. More specifically, results show that the use of bare-metal virtualization can satisfy the performance demands and deliver performance predictability for HPC applications.

3.2 Related Work

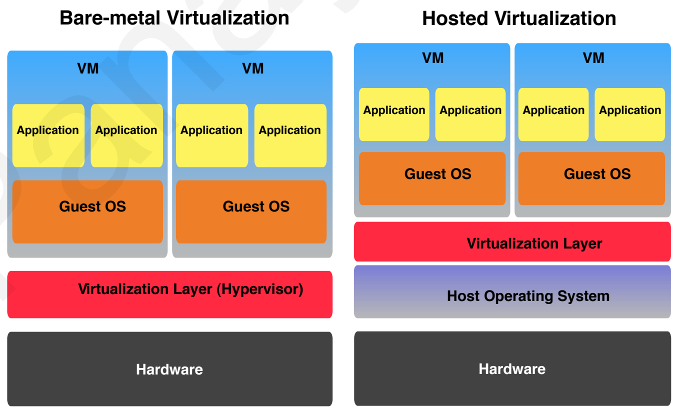
Using virtual machines is not a new trend and it has not been ignored by the high performance computing society. Because of the benefits they offer, there is a large interest in using virtual machines for HPC despite the performance penalty that is caused by the additional virtualization layer.

Macdonell and Lu [32] state that VMs is a good solution to abstract out the heterogeneity in order to fully utilize metacomputers and grids. Although the use of VMs has overheads, recent improvements in software and hardware support reduce the overheads for HPC applications. On their work the authors show a simple quantitative study of the overheads of running the benchmarks BLAST, HMMer and GROMACS under VMWare. In the paper they support that while not perfect, VMs are emerging as a pragmatic tool in HPC. Menon *et al.* [33] used a system-wide statistical profiling toolkit implemented for the Xen virtual machine environment in order to analyze the performance overheads incurred by networking applications running in Xen VMs and their work was focused on networking applications. Soltesz *et al.* [34] presented an alternative approach to hypervisors which suite better for HPC clusters. Their approach was a synthesis of resource containers and security containers applied to general-purpose time-shared operating systems. Their results showed the benefits that HPC clusters can retrieve from these techniques and therefore achieving efficiency and isolation. The work by Youseff *et al.* [35] states that despite of the potential benefits, performance advances, and recent research indicating its potential, virtualization is currently not used in high-performance computing (HPC) environments. As they say one reason for this is the perception that the remaining overhead that VMMs introduce is unacceptable for performance-critical applications and systems. The authors conclude that Xen the paravirtualizing system poses no statistically significant overhead over other OS configurations currently in use at LLNL for HPC clusters. Huang *et al.* [36] present a case for HPC with virtual machines by introducing a framework which addresses the performance and management overhead associated with VMbased computing.

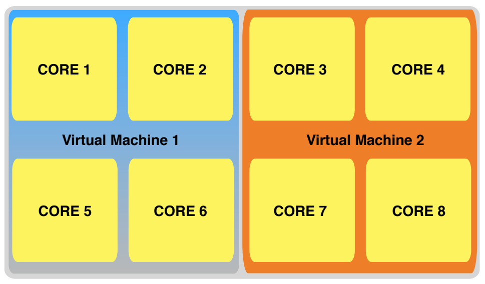
Two key ideas in their design are: Virtual Machine Monitor (VMM) bypass I/O and scalable VM image management. Rodríguez *et al.* [27] propose to achieve better management of physical resources by dynamically reallocating and adjusting local resources according to demand. They present and evaluate a component to provide dynamic adjusting of CPU resources in a multi-core VM-based resource provider. Matthews *et al.* [26] showed that in an operating system level virtualization system, applications suffer from high execution time variance compared to other virtualization methods. Payer *et al.* [37] proposed extending the Xen Hypervisor scheduler in order to isolate the systems' CPU resources from the additional scheduling jitter introduced by the Linux kernel scheduler.

Overall, many research works have focused on the use of virtualization for the execution of HPC applications. Most of these works are based on the Xen Hypervisor and also on the design of efficient communication between multiple processors. In this work virtualization solutions for a multi-core processor [10] are studied. More specifically, this work is focused on performance isolation for HPC applications showing that without any modifications on the system or applications high levels of performance can be achieved.

3.3 Virtual Machines, HPC and Performance Domains on Multi-core Processors



(a)



(b)

Figure 3.1: (a) Virtualization techniques bare-metal and hosted and (b) performance domains on a multi-core system.

In general, virtualization is used to abstract the system from the physical resources that are actually present. There are mainly two ways of implementing it (see Figure 3.1 (a)): Hosted Virtualization, and Bare-metal Virtualization. The former is usually implemented on top of a Host Operating System while the latter uses a hypervisor layer directly on top of the hardware. Bare-metal virtualization provides some interesting features, for example besides providing memory management, and CPU scheduling, it considerably reduces the interpretation overheads using fewer resources, and it allows the OS kernels to work on top of it as-is. A hypervisor is a layer of software, more like a modified Linux kernel, which runs directly on a systems hardware. As such, it is like replacing the operating system and thereby allowing to run multiple operating systems concurrently as if they had their own hardware without having extra layers between them. In this chapter the use of virtualization in multi-core systems is studied, and more specifically their use for High Performance Computing (HPC) maintaining high levels of performance and performance isolation.

With the use of virtualization it is possible to achieve isolation among the execution of different applications running at the same time on the same multi-core. This isolation means that it is possible to split the multi-core into different *Performance Domains*. Consequently, an application running within such a domain receives all the resources assigned to that domain. The *Performance Domains* can be exploited to execute HPC applications that will achieve a speedup close to native execution. The relevant advantage is the fact that the performance achieved by that application on that domain is isolated from the execution of any other applications on other domains and therefore limiting their interference compared to a native system. Figure 3.1 (b) depicts a multi-core processor where the cores are grouped into different Virtual Machines, which in turn implement the *Performance Domains*. Different roles for each Domain can be assigned. For example one domain could be dedicated for HPC application execution and other domains for interactive job execution and software development and testing.

3.4 Experimental Setup

For the purpose of this work two sets of experiments were performed. The objectives of these experiments are first to investigate the different configurations of VMs in terms of assigned cores per VM, and second to study the performance interference between different application executing concurrently and the use of virtual machines to provide performance isolation.

Table 3.1: PARSEC Application Description.

Application	*Brief Description*
Blackscholes	Performs option pricing using the Black-Scholes partial differential equation (PDE) method
Bodytrack	Performs the tracking of people in security camera images
Facesim	Simulates the motions of a human face
Fluidanimate	Models the fluid dynamics for animation purposes using the Smoothed Partical Hydrodynamics (SPH) method
Raytrace	Renders a 2D image out of a 3D model using the ray-tracing method

For all experiments an 8-core computer system equipped with two 4-core Intel(R) Xeon(R) CPU E5320 processors running at 1.86GHz is used. These processors are configured with 128KB of private L1 cache and 8MB of shared L2 cache. This system was running the 64-bit version of Ubuntu 9.04. This is what is called in this work the *native* system although in the virtualization terminology this is known as the *host* system.

The hosted virtualization for this work was implemented using the Virtual Box package from Oracle [30] version 3.1. VirtualBox is a collection of powerful virtual machine tools, targeting desktop computers, enterprise servers and embedded systems. With VirtualBox, you can virtualize 32-bit and 64-bit operating systems on machines with Intel and AMD processors, either by using hardware virtualization features provided by these processors or by software [30]. For bare-metal virtualization the Xen hypervisor version 4.0.2 [31] is used. After producing the image of the Virtual Machine using Virtual Box and Xen the resources that could be used are allocated to the machine. Either only Virtual Box VMs or XenVMs are used for each set of experiments but with the same system configurations. The memory allocated to the VM was 4GB of RAM. The number of processors allocated was either 2 or 4 cores for the virtual machine resulting in two different virtual machines: *VM2* and *VM4*. In order to have fair comparisons, VM had the same OS installed as the one for the native machine, the Ubuntu 9.04 OS, which in the virtualization terminology is known as the *guest* system.

The workload used for the experiments of this work consists of five applications from the PARSEC benchmark suite [29]. PARSEC is a benchmark suite composed of multithreaded programs that focus on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors. The applications come from many different areas such as computer vision, video encoding, financial analytics, animation physics and image processing. The five applications used in this work were selected in order to represent different categories of applications with different characteristics on their execution and they are: *Blackscholes*, *Bodytrack*, *Facesim*, *Fluidanimate*, and *Raytrace*. Their brief description is presented in Table 3.1. Applications were selected in order to have representative workloads of applications with different characteristics and behaviour during execution [29]. More specifically the characteristics of the selected applications are: *(i) Blackscholes*, has a small number of read and write operations, which will not suppress the shared resources of the system, *(ii) Facesim* and *Raytrace* are applications with large number of read and write operations and high FPOS and are selected in order to study how they interfere their execution with other applications on the shared resources of the system, and *(iii) Bodytrack* was selected according to its characteristics of high reads and low writes but with high number on synchronization operations (locks and barriers) to study its behaviour during co-execution with other applications. It can be seen from the selected applications that they have different characteristics in terms of both memory and resource requirements.With such a set it is possible to ensure that representative workloads of applications of different characteristics are selected and at the same time to study how their co-execution influences each applications' performance.

The PARSEC applications used have been parallelized with POSIX threads (*pthreads*). For the presented experiments the applications are compiled using *parsecmgmt* which is the main tool that comes with PARSEC to build and run packages. The configuration *gcc-pthreads* is used as to build the parallel executable of the applications. The compiler used was *gcc-3.4* and *g++* compiler. When running the PARSEC applications he *native* input data set is used which is the input intended for large-scale experiments of performance measurements and research.

The scenarios of the applications concurrently executing on the different VMs are

Table 3.2: Experiments scenarios for 2 and 4 instances of virtual machines.

Scenarios for 2 Instances of VMs	
Scenario	*Concurrent Applications Execution*
Scenario 1	Bodytrack, Blackscholes
Scenario 2	Bodytrack, Raytrace
Scenario 3	Bodytrack, Fluidanimate
Scenario 4	Bodytrack, Facesim
Scenario 5	Blackscholes, Raytrace
Scenario 6	Blackscholes, Fluidanimate
Scenario 7	Blackscholes, Facesim
Scenarios for 4 Instances of VMs	
Scenario	*Concurrent Applications Execution*
Scenario 8	Blackscholes, Bodytrack, Raytrace, Fluidanimate
Scenario 9	Blackscholes, Bodytrack, Raytrace, Facesim
Scenario 10	Blackscholes, Bodytrack, Fluidanimate, Facesim
Scenario 11	Bodytrack, Raytrace, Fluidanimate, Facesim

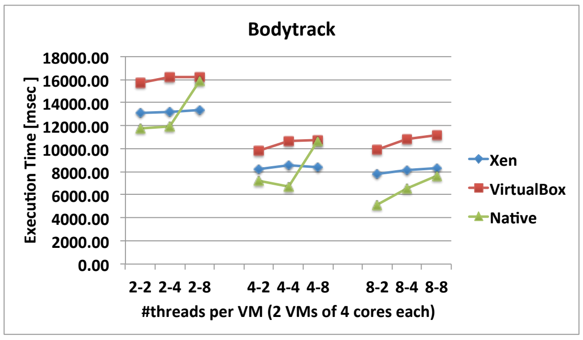
depicted in Table 3.2. For the first set of experiments, 2 Virtual Machines instances of 4 cores each are created. For the second set of experiments 4 Virtual Machines instances of 2 cores each are used. Furthermore, the number of parallel threads of each application was varied from 2 to 4.

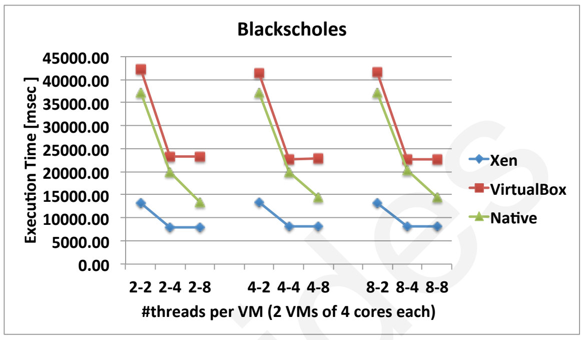
The measurement of the execution time was performed using the *time* shell command of Linux on both the native and the VM systems. The start of the applications execution was concurrent to all VMs. In the case of applications with different execution times, the ones with the smallest time of execution were re-executed to reach the time of execution of longest applications without considering their extra execution in the results measurements. This assumption was made in order to have uniform interference of applications executed in the system. If otherwise mentioned, the execution time and other metrics refer to the *real* or total execution time reported by the *time* command. As for the execution time on the VMs, the time measurement was validated using a different process where on the VM (*guest* system) a simple client program requests the time to be measured by a simple server on the native or *host* system. The differences between the two methods were negligible.

3.5 Experimental Results

For the first set of experiments 2 VMs with 4 cores each are used. As described in Section 2.6 7 Scenarios are selected for execution. Those Scenarios are executed on Virtual Box, Xen hypervisor and Native and with number of threads for each application varying from 2 to 4. In Figures 3.2 (a) and (b) the results from the Scenario 1 are presented and in Figures 3.3 (e) and (f) the results for Scenario 6. In Figures 3.2 and 3.3, the y-axis shows the execution time on the different VMs and the x-axis shows the number of threads per application in the order described in Table 3.2. For example, in Scenario 1 and 2-2 setup, both applications are executed with 2 parallel threads each whereas in setup 2-4 *Bodytrack* is executed with 2 parallel threads and *Blackscholes* with 4 parallel threads.

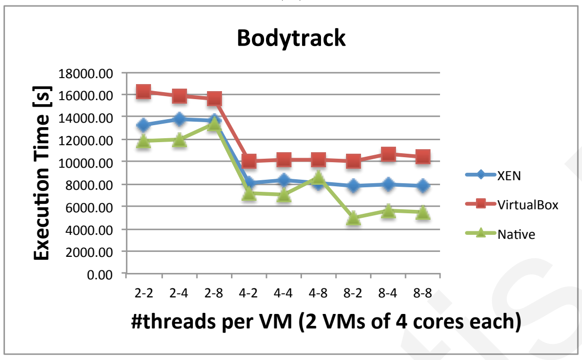
From the results it is possible to observe that for all applications and both scenarios, when increasing the number of threads from 2 to 4, it is possible to observe the reduction in the execution time as expected. The degree of reduction is determined by the characteristics of the applications as mentioned later. For the execution with 8 threads, it is possible to observe different behaviors for the native and virtualized

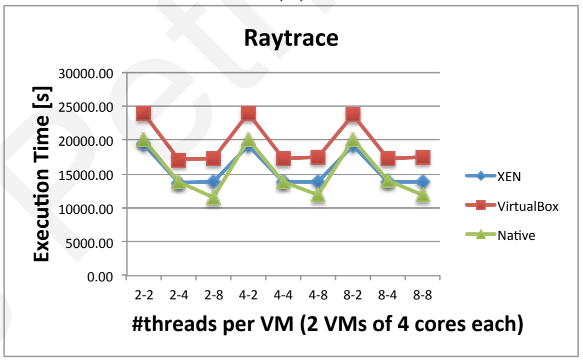




(a)

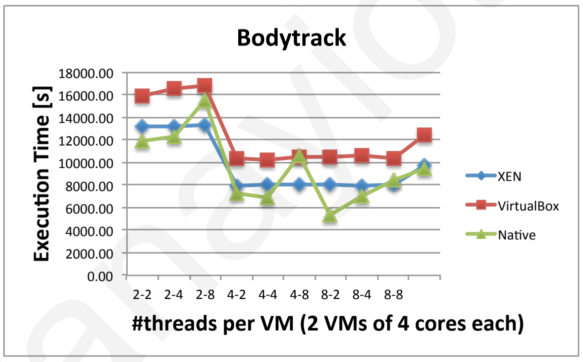
(b)

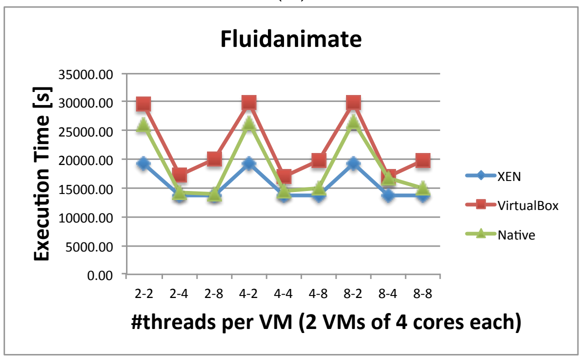




(c)

(d)

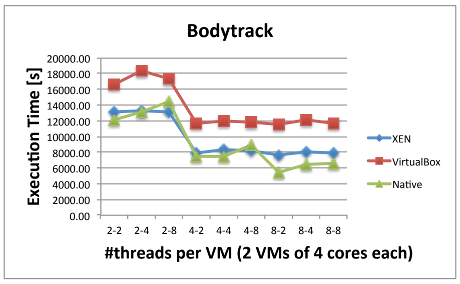




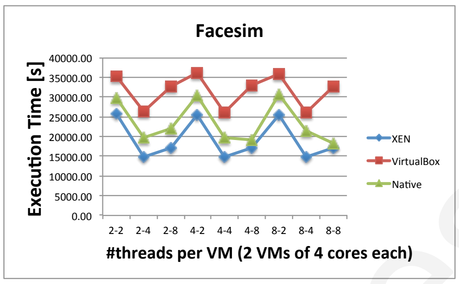
(e)

(f)

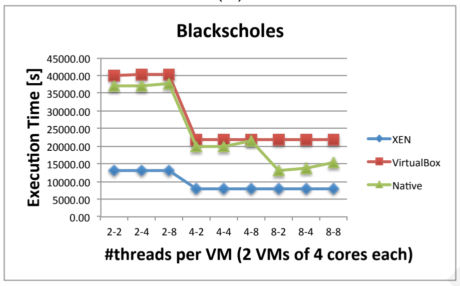
Figure 3.2: Execution Variance on VirtualBox Native and Xen for: Scenario 1 (a) and (b), Scenario 2 (c) and (d), Scenario 3 (e) and (f).



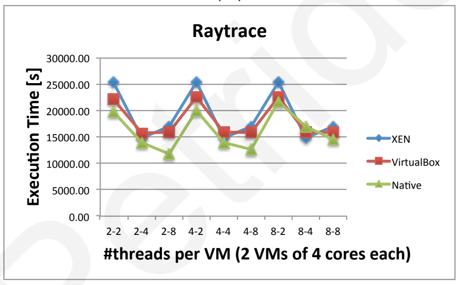
(a)



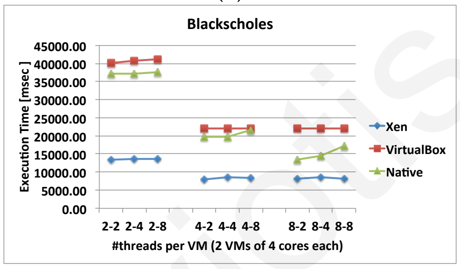
(b)



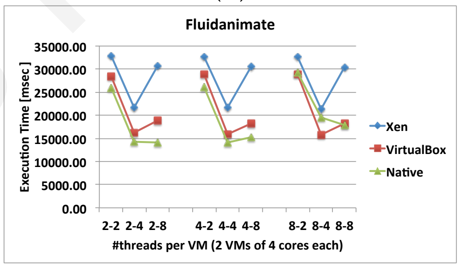
(c)



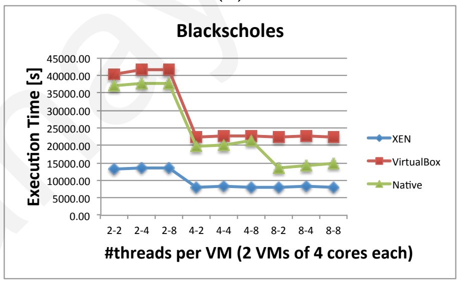
(d)



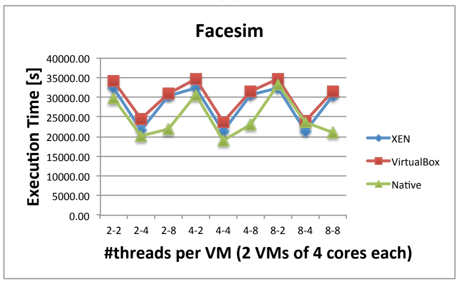
(e)



(f)



(g)



(h)

Figure 3.3: Execution Variance on VirtualBox Native and Xen for: Scenario 4 (a) and (b), Scenario 5 (c) and (d), Scenario 6 (e) and (f), Scenario 7 (g) and (h).

environments. For the virtualized environments both Virtual Box and Xen provide the same performance for 8 threads as it was for 4. This is due to the fact that within the VM, the execution is limited to the available resources to the VM which in this case is 4 cores. As for the native execution, while for Scenario 1 *Blackscholes* shows a further reduction of the execution time from 4 to 8 threads, for Scenario 6 *Fluidanimate* shows an increase in the execution time from 4 to 8 threads. Notice that in both cases the changes in performance for the 8 threads create an interference with the reference application *Bodytrack* and *Blackscholes* for the Scenarios 1 and 6 respectively. The reason for this different behavior has to do with the characteristics of the combined applications. For example, *Bodytrack* does seem to allow *Blackscholes* to "steal" cores in Scenario 1 while 8 threads of both *Blackscholes* and *Fluidanimate* seem to result in a thrashing of the system as both applications' performance suffer considerably.

For the second set of this work experiments 4 VMs with 2 cores each are used. As described in Section 2.6, 4 Scenarios are selected for execution. Those scenarios are executed on Virtual Box, Xen hypervisor and Native and with number of threads per application varying from 2 to 4 with all possible combinations for the different applications. In Figures 3.4, 3.5, 3.6 and 3.7 the y-axis shows the execution time on the different VMs and the x-axis shows the number of threads per application in the order described in Table 3.2. For example, in Scenario 8 and 2-2-2-2 setup all applications are executed having 2 parallel threads each whereas setup 2-2-2-4 *Blackscholes*, *Bodytrack*, *Raytrace* are executed with 2 parallel threads and *Fluidanimate* with 4 parallel threads.

The results depicted in Figure 3.4 show the execution time of the applications on the different environments (VirtualBox, Xen and native) for the different setups. The first important observation from the depicted results is that the execution time in the Virtualized environments seems very stable. Notice that a virtual machines is running with only 2 cores each and that the executions are from 2 to 4 threads. Therefore, it is expected that the performance does not change for the different setups.

It can be observed the small execution variance between the virtualized environments and the native execution. These results justify the performance isolation offered by the virtualization environments. Also from the experiments, it is obvious that both VirtualBox and Xen add an extra overhead on the applications execution compared to the native. This overhead ranges from to 10 to 37% of additional overhead for VirtualBox and from 1 to 3% for Xen. These results shows that a bare-metal virtualization system can deliver the performance very close to the performance on the native system.

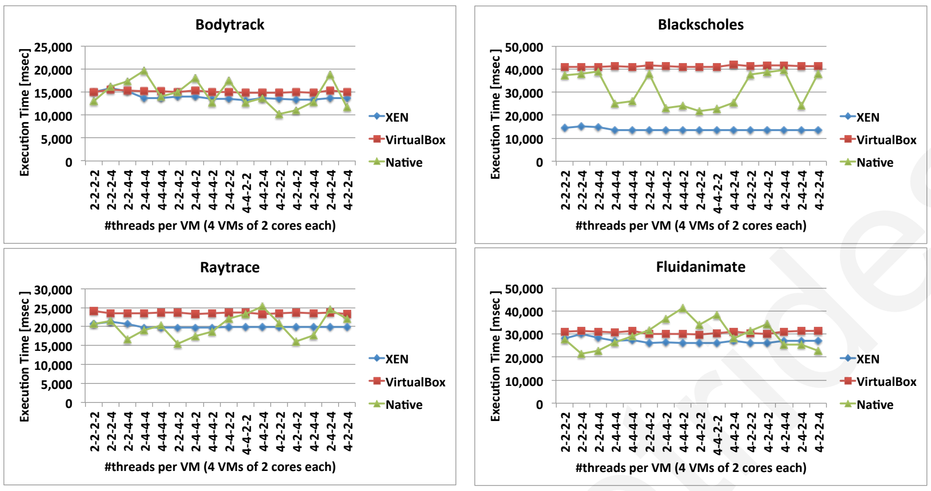


Figure 3.4: Execution Variance on VirtualBox Native and Xen for Scenario 8.

The different behavior observed from the experimental results can be explained by the characteristics of the applications executed. These applications are classified according to their characteristics. In the first class belong applications that are heavy computational and that use little data. As the execution of the virtualized code is performed natively on the system these applications are expected to suffer only a small performance overhead from virtualization. One such application is *BlackScholes* which shows better performance very close to the native execution. The second class includes applications that handle large input data sets and as such their overhead is slightly larger. In this class both *Fluidanimate* and *Raytrace* can fit. Finally, in the last class includes applications that suffer from the virtualized environment. In the case of such workload *Bodytrack* is such application. The overhead observed for *Bodytrack* application is mainly due to the high number of barriers, *i.e.* lock- and barrier-based synchronizations, and the high number of waits of condition variables, in contrast to the other applications [29].

Studying the results depicted in Figures 3.2, 3.3, 3.4, 3.5, 3.6 and 3.7 closer it can observed that there are cases where the performance on the virtualized system is better than on the native system. As explained before, this is due to the fact that in a native system the co-execution of applications may result in contention in the

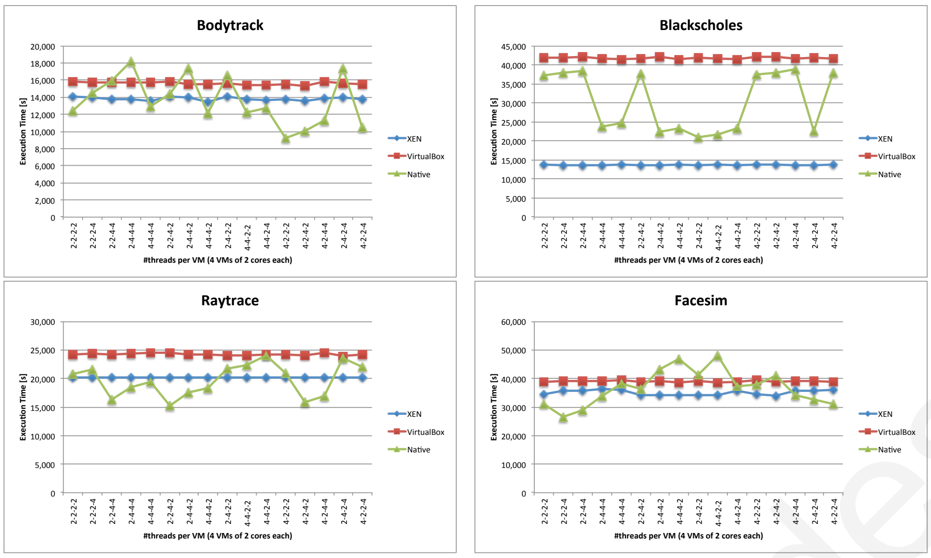


Figure 3.5: Execution Variance on VirtualBox Native and Xen for Scenario 9.

resources and thus a performance degradation. A simple analysis of the results is presented in Figure 3.8. When considering the virtualization with Virtual Box, for 19% of the experiments the execution on the virtualized environment achieved a better performance than the native execution. As for the virtualization with Xen, this value increased to 60%, *i.e.* for 60%, of the experiments the execution in the Xen, environment achieved a better performance than the native execution.

Overall the experimental results show that the execution on virtualized environments (specially when using Xen) offers performance isolation at a negligible cost in terms of performance overhead. This performance isolation allows for applications to achieve their predicted speedup without suffering any interference from co-execution of other applications on the same processor at the same time. This is becoming more relevant for large-scale multi-core processors which should be kept fully utilized by running simultaneously different applications.

3.6 Summary

Virtualization is a technique that offers many benefits such as hiding the hardware complexities or creating different images of the hardware as to isolate users or install different systems. One disadvantage of hosted virtualization is the fact that the introduction of an intermediate layer between the hardware and the operating system results in performance penalty. This technique was revived recently as

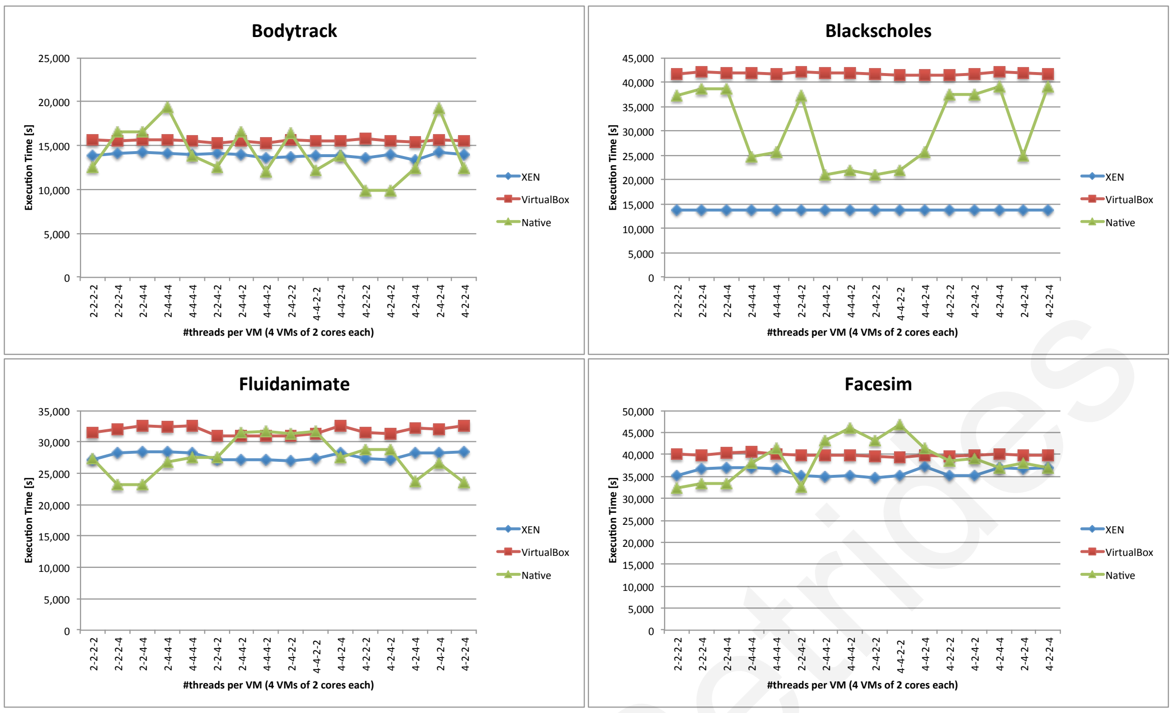


Figure 3.6: Execution Variance on VirtualBox Native and Xen for Scenario 10.

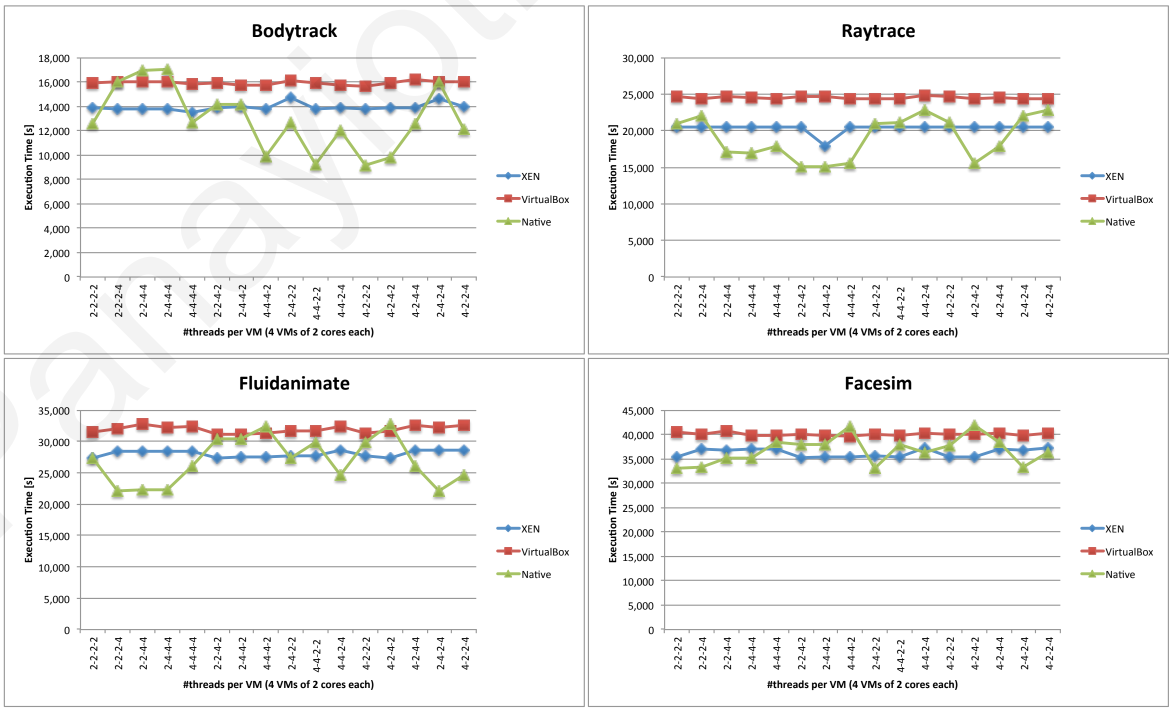
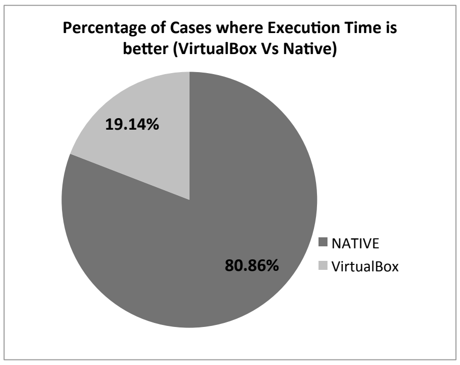
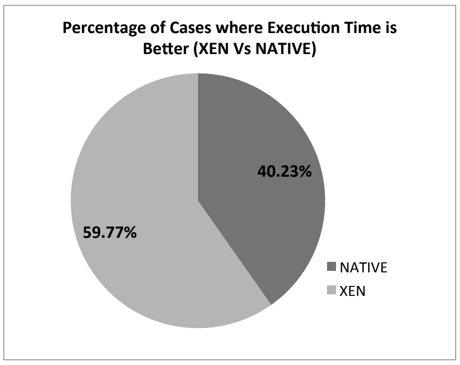


Figure 3.7: Execution Variance on VirtualBox Native and Xen for Scenario 11.



(a)



(b)

Figure 3.8: Performance comparison between: (a) VirtualBox and Native Execution and (b) Xen and Native Execution.

to improve the utilization in large data-center systems. The limitations of hosted virtualization are minimized with the use of bare-metal virtualization and from the results it is obvious that applications' performance may benefit from the use of this technique. With the availability of processors with increasing number of cores and increased complexity, virtualization will soon become present in all systems. As such, it is important to understand the benefits that can be obtained for multi-core systems as well as the impact of the different kinds of virtualization on applications and specially on the most demanding HPC applications, as they require to exploit all available performance.

This work analyzes the performance overhead and performance isolation of parallel applications while executing on top of different virtualization environments on a 8-core based system. The results of the experiments show that for most applications the overhead is relatively small. Virtualization shows to be an important tool as to create performance domains where the performance of HPC applications can be safeguard, independent of the applications executing on the rest of the multi-core processor. Also, it is important to mention that by using vitualization not only predictable performance is achieved but also the interference of applications co-executing is limited on the virtualized system compared to their co-execution on the native system. It can be also observed that for more than half of the presented experiments the execution on the bare-metal virtualized system achieved better performance than the native execution. Moreover, the overhead observed for the execution of the applications was only up to 3%. Performance isolation provided by virtualization, allows applications to achieve their predicted speedup without suffering any interference from co-execution of other applications on the same processor at the same time. This is becoming more relevant for large-scale multi-core processors on which full utilization is desired by running simultaneously different applications. Overall the results were very encouraging for the use of virtualization for future large-scale multi-core processors even for demanding HPC applications.

Chapter 4

Heterogeneous and NUMA-aware Scheduling

As the number of cores increases in a single chip processor, several challenges arise: wire delays, contention for out-of-chip accesses, and core heterogeneity. In order to address these issues and the applications demands, future large-scale many-core processors are expected to be organized as a collection of NUMA clusters of heterogeneous cores. In this chapter a scheduler is proposed that takes into account the non-uniform memory latency, the heterogeneity of the cores, and the contention to the memory controller to find the best matching core for the application's memory and compute requirements. Scheduler decisions are based on an on-line classification process that determines applications requirements either as memory- or compute-bound. This work is evaluated on both a real clustered many-core architecture, the 48-core Intel SCC [1], and its adaptability and scalability using a simulated clustered many-core architecture. On both cases applications from the SPEC CPU2006 benchmark suite were used. Results show that even when all cores are busy, migrating processes to cores that match better the requirements of applications results in overall performance improvement. In particular a reduction of the execution time from 15% to 36% is observed compared to a random static scheduling policy. In addition, the effectiveness and adaptability of the proposed approach is evaluated using the Sniper simulator [38]. Results show that as the number of cores within a cluster increases, the proposed scheduling policy can still reduce execution time of applications of up to 30% for compute-bound applications and up to 16% for memory-bound applications.

4.1 Motivation

The current *de-facto* standard in processor design is the multi-core architecture which emerged as a way to provide at the same time energy efficiency and increasing compute power. Moreover, as technology and on-chip integration keeps evolving, the number of cores per chip tends to increase. At the same time, processors start integrating cores of different characteristics and include features to change their characteristics dynamically at runtime as to improve the efficiency of hardware. While the increasing number of cores may result in a larger degree of parallelism, it may not necessarily lead to improved performance. Considering processors evolution, they are becoming very complex systems and as the number of cores increases some features make the execution non-uniform across different cores. In this work three factors that affect the non-uniformity of applications execution are identified: *(1)* non-uniform memory latency due to variable number of hops from the core to the memory controller; *(2)* non-uniform execution due to cores with different characteristics (heterogeneous or different operation modes of identical cores); *(3)* non-uniform memory latency due to contention on the access to a shared memory controller.

The target architecture for this work is one where the many-core processor is composed of different clusters of cores, each one served by a different memory controller. The different distances of cores to the memory controller within a cluster determine the non-uniform latency to memory. In addition, it is assumed that cores are heterogeneous, in this work heterogeneity is emulated by having cores execute at different operating modes (low-power versus high-performance). A cluster of such an architecture is depicted in Figure 4.1.

The key for determining the best matching core for a certain application is to find out the application's memory and computational requirements. As it is assumed no previous knowledge of the application, whenever an application enters the system, the scheduler assigns it to different cores as to acquire enough data to perform the classification. It is important to classify the application as memory- or compute-bound so it is given the best matching core. In addition, it is necessary to have an order among different applications of the same category. Priority should be given to the applications with the highest requirements. For example, the applications with the highest memory requirements should be placed on the cores closest to the

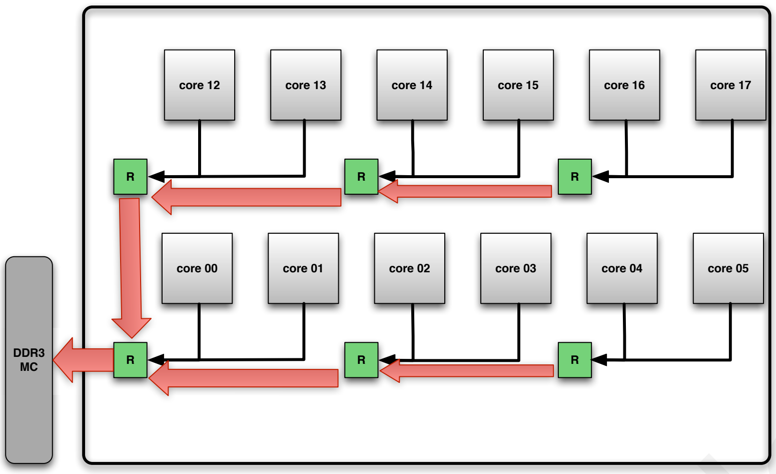


Figure 4.1: Cluster of a Many-Core Processor. The arrows show the distance of a core to the memory controller and their thickness represent the accumulated bandwidth on the links.

memory controller.

After the classification and the ordering within each category, the scheduler uses a heuristic to place the application in the best matching core. If such a core is not free, then it considers the overhead of displacing the application from that core versus the cost of assigning the incoming application to an alternative non-optimal matching core. In addition to the placement, the scheduler monitors constantly the behaviour of the applications. If changes are observed, for example resulting from an application entering a new phase, a classification phase is triggered in order to determine a new better placement. Finally, it should be mentioned that since classification of applications is applied on a live system with applications co-executing, indirectly it is observed how the performance is affected from the interference on shared resources.

Given the application, determining appropriate resources for power-performance efficiency is not a trivial task. In addition, given the multiple goals that the proposal tries to satisfy it is very difficult in practice to coordinate the scheduling operations. This work propose a scalable heuristic dynamic scheduling policy, which tries to satisfy applications requirements in terms of computation and memory during their execution. The impact of the different factors affecting applications performance is quantified from specific performance penalty functions using the gathered metrics.

The proposed scheduler is evaluated on a 48-core Intel SCC [1] with applications from SPEC and NAS benchmark suites. Applications are classified based on their computation and memory requirements. This classification is a result of studying the execution of applications and gathering the required information, such as IPC, off-chip memory requests and last level cache misses among others. The impact of each factor is quantified and used by the proposed scheduler in order to improve applications performance. From the experimental results it is shown that the proposed scheduling balances very well the power-performance efficiency of the system and in addition improves their performance, compared to an agnostic static assignment policy. In particular, it is observed a reduction of the execution time up to 36% for the compute-bound applications and up to 15% for memory-bound applications when compared to a random static scheduling policy.

Moreover, to study the proposed scheduler effectiveness and adaptability a clustered many-core architecture is simulated using the Sniper simulator [38]. The objectives of this simulation were as the number of cores within a cluster increases to explore: *(i)* the effectiveness of the proposed scheduler; *(ii)* each scheduling decision factor impact to applications performance; *(iii)* the proposed scheduler adaptability on different ratios of cores with different characteristics in terms of core frequency.

The main contribution of this work is:

- Propose a dynamic on-line classification methodology by determining the degree of memory- and compute-bound for each application.
- Propose and implement a scalable dynamic scheduling policy for future heterogeneous many-core architectures.
- Evaluate the scheduler using applications from SPEC benchmark suite on the 48-core Intel SCC processor.
- Evaluate the effectiveness and adaptability of the proposal on larger simulated clustered many-core architectures.

4.2 Related Work

Characterizing applications behaviour has been studied from different researchers over the years. In addition, focusing on scheduling and resource management based on applications' behaviour and systems resources has been presented over the years targeting high-performance computing systems, improving applications performance, resource utilization among others. In this section the work most relevant to this one is presented, particularly focusing on policies and methods used in order to address challenges raised from clustered and multi-core systems.

Rogers *et al.* [39] showed that the scalability of multi-core architectures is limited from the off-chip bandwidth. Studies focused on how to reduce memory latency using different scheduling techniques. Awasthi *et al.* [40] showed how an effective dynamic page migration policy can reduce memory latency and improve a multi Memory Controller system throughput. Characterizing applications in terms of computation power and memory requirements was the target of different works such as [41]. Long *et al.* [42] tried to characterize different applications from SPEC benchmark suite according to their bandwidth requirements. Winter *et al.* [43] used the Hungarian algorithm to characterize applications and to determine which resource is the most appropriate but their interference is assumed limited.

One of the first approaches on resource management is batch scheduling. In this approach the resource allocation is the users responsibility to specify the priority of each job and it is stored in queues on which the job with the highest priority is scheduled first [44–46]. On the same context is the modern cluster resource management, where users must specify their resource requirements. Fairness policies can be adopted from this techniques in order to monitor jobs resource requirements and resource availability and utilization [47,48]. Model-based scheduling [49,50] is another approach of addressing scheduling for resource management and applications performance. One proposal is the use of utility functions [49]. These functions are derived from off-line measurements of raw resource utilization describing applications execution on the system environment and are used to optimize both resource utilization and applications execution. Feedback-driven techniques are also adopted by this approach in order to introduce reinforcement learning and optimize utility functions. Also, off-line workloads models can be used in order to optimize utility functions [51–53].

Most recent work focused on how applications interference on shared resources, such as caches, influences their performance. Different approaches were proposed in order to address this issue with most of them focusing on the interference of applications on a shared cache [54]. The target of such works is to quantify or predict the interference between co-scheduled applications. One approach of addressing this issue is the disjoint resource utilization where applications with disjoint resource requirements are co-scheduled in order to minimize interference. This approach is achieved by either using hardware measurements information or using working sets sizes to make co-scheduling decisions. Another approach of addressing this issue is by using interference experiments [53,55]. This approach uses on-line experiments with different combinations of applications and the highest performing combination is selected. Predicting interference is another approach of addressing this issue [56–59]. In this approach, past measurements or performance models to predict the expected interference between applications is used in order to take scheduling decisions. Most of these works are focusing on the slowdown from cache effects or analytical models to predict cache misses for co-scheduled applications [60–62]. Fedorova *et al.* [63] suggested that the Operating System should handle scheduling of threads by monitoring physical variables [64]. Shelepov and Fedorova [65,66] address the scheduling in heterogeneous multi-core systems in the case of short-lived threads, which do not allow monitoring to reach a near optimal scheduling solution. They schedule threads according to their architectural signatures, which are composed of certain microarchitecture-independent characteristics, generated offline relying on the developer that a thread will exhibit a typical behaviour during the generation of the signature. If that case cannot be ensured, the signature should be generated through several runs and their results combined to obtain the final signature. Li *et al.* [67] focused on load balancing to increase performance using a NUMA-aware scheduling technique. Kumar *et al.* [68,69] focus on developing algorithms to schedule applications on cores that best match their execution requirements for two types of cores on a small exploration space. Haritatos *et al.* [70] suggested a co-scheduling approach for CMPs by monitoring applications interference on shared resources and focused on utilizing the entire memory of CMPs. Heirman *et al.* [71] proposed a scheduling policy that dynamically matches applications' working set size and off-chip bandwidth requirements with the available off-chip bandwidth proposing hardware adjustments.

Kaliorakis *et al.* [72] studied error detection on a many-core system using functional online error detection methodology in many-core architectures. Their approach was to accelerate the online error detection methodology but at the same time reduce the duration of the test programs executed and limiting the contention of cores to shared resources. In this work they used both memory- and compute-intensive workloads. Their findings show that memory-intensive workloads are affected the most by the excessive traffic in the interconnection network and the DRAM controllers. Their approach to overcome this challenge, and therefore improve the performance of such workloads, is a proposed parallelization method which utilizes both private memory of the cores and the shared on-chip MPB memory. Their results show that they succeed to minimize the effect of both access latencies and the traffic to the DRAM controllers and therefore accelerating the performance of memory-intensive workloads.

This work, compared to the related work presented above, differs in the following main aspects. First, it targets and evaluates many-core architectures. It is shown that two main factors affect applications execution in such architectures. More specifically, the first factor is the distance of a core to the memory controller and the second factor is the cores' computational capabilities. Secondly, this work tries to tackle these factors by proposing an on-line scheduling policy, which has no previous knowledge of applications' execution behaviour on a many-core architecture and in addition with no user interference or off-line profiling. Applications are classified either as memory- or compute-bound by evaluating the respective factors of each application at runtime. A methodology for dynamic on-line classification of applications using performance penalty functions is proposed, which can determine the impact of each factor for each application, and can predict its impact to other system resources. The on-line scalable heuristic scheduling policy manages to minimize the performance impact of the above mentioned factors and at the same time increase system throughput. In addition, the proposed scheduler manages to quantify application requirements and tries to satisfy them at runtime.

4.3 Challenges of Clustered Many-Core Architectures

In this section, the architecture of scalable many-core systems is described, which are the target of this work, and focus on the challenges for such architectures.

4.3.1 Clustered Many-Core Architectures

Clustered many-core architectures consist of tiles of cores with private L1 and L2 cache, interconnected by a 2D-grid network. Off-chip memory requests are served

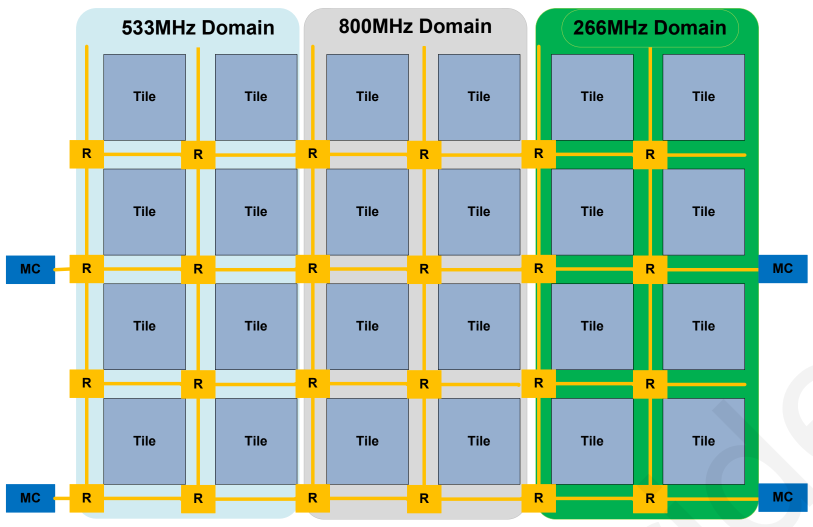


Figure 4.2: Core Clock Frequency System Configuration

by a number of memory controllers which are dedicated to a cluster of cores [1].

In such architectures specific factors should be studied to define their impact to applications performance and consequently system throughput. More specifically, as the number of cores per cluster increases, so does their distance to the memory controller. This factor leads to non-uniform memory accesses (NUMA) and consequently influences the application execution time.

Future clustered many-core architectures may consist of resources of different computational capabilities. This configuration can result on different performance and power efficient domains, which can satisfy different application requirements and therefore increasing system throughput and power efficiency. In this work heterogeneity is emulated by defining domains of cores of different operation modes (low power, standard and high-performance) as depicted in Figure 4.2.

It can also be considered that in such architectures, the number of cores per memory controller will increase faster than the number of on-chip memory controllers. This is a consequence of the available area and the off-chip pins that do not allow the increase on the number of on-chip memory controllers. This effect becomes more pronounced as the cumulative off-chip bandwidth requests of co-executing applications saturate the off-chip bandwidth.

Examples of clustered many-core architectures are the Intel SCC [1] and Tilera

[73] processors. For the purpose of this work the 48-core Intel SCC processor is used as a case study. Intel SCC has dual-core tiles interconnected by a 2D-grid network. A mesh interface unit (MIU) connects the tiles to a router of the network. The addresses corresponding to this memory are mapped through a single memory controller. Off-chip memory requests are served by a number of memory controllers which are dedicated for each cluster (12 cores per memory controller).

4.3.2 Non-Uniform Memory Latency

The first factor studied on a clustered many-core architecture is how the distance of a core to the memory controller can influence applications performance. For the purpose of this work, it is considered that each memory controller is responsible for a cluster of cores, as the configuration of the Intel SCC processor. As depicted in Figure 4.1 the number of routers (R) that are involved in the route from a core to the memory controller are considered as hops (H). The further a core is, the more hops are involved to its route to the memory controller and therefore its distance is higher. This distance affects the delay of off-chip memory requests and therefore this factor becomes more serious as the number of cores per cluster increases. The arrows depicted in Figure 4.1 show the route that memory requests take in order to reach the memory controller that serves the corresponding core.

Figure 4.3 depicts applications behaviour as the distance of a core to the memory controller increases. The first observation is that some applications are influenced more from this distance than others. Applications affected the most are memory-bound and those are: *dc*, *bt*, *lu*, *sp*, *ua*, *sphinx* and *libquantum*. Compute-bound applications, such as *ep* and *povray*, do not show significant performance degradation due to this factor.

The second observation is that, application overheads change in a linear way and therefore it can be predicted for each application at each distance. In order to justify these findings applications behaviour is classified in terms of distance using the IPC metric at each execution point as presented in Figure 4.3 (trend line of model graph).

$$\begin{aligned} D(H) &= a \times H, \\ a &= \frac{IPC_x - IPC_y}{H_x - H_y} \end{aligned} \tag{4.1}$$

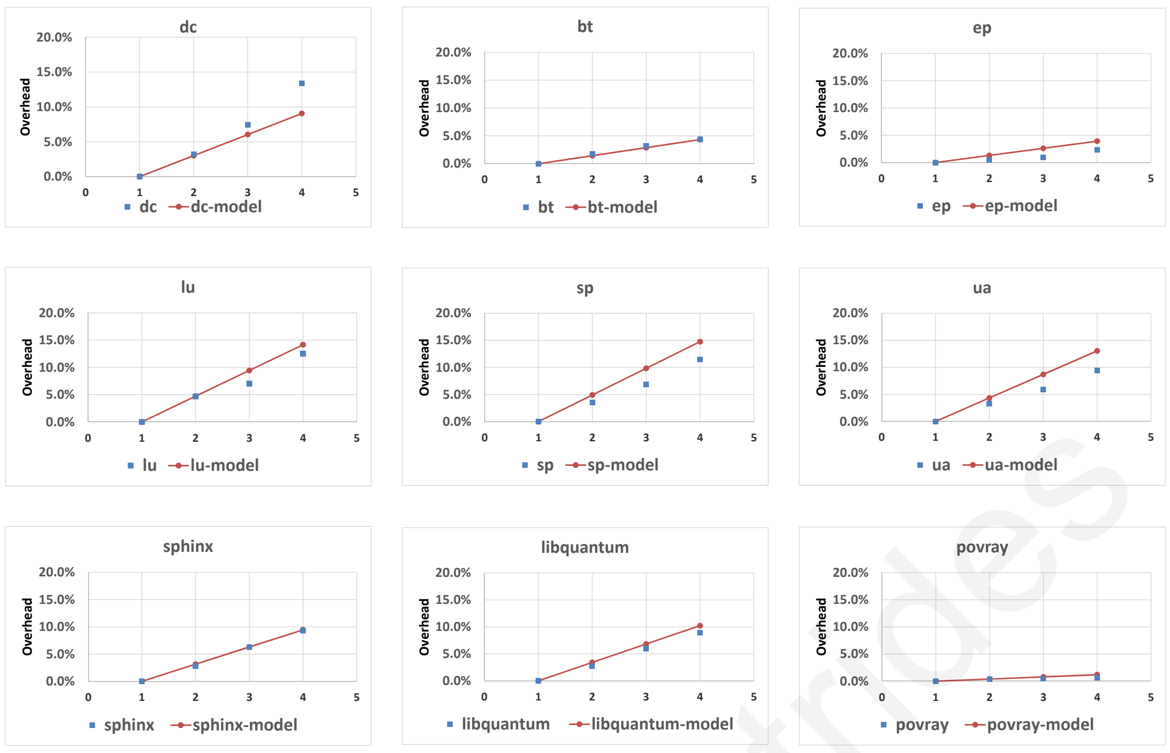


Figure 4.3: Core to Memory Controller Distance Model Execution Performance Influence for SPEC and NAS applications.

As it could be considered that in such architectures the number of cores per memory controller will increase faster than the number of on-chip memory controllers, the same experiments have been executed on a simulated many-core architecture using the Sniper simulator [38]. More specifically, the distance of a core to the memory controller is increased up to 10 in order to study its impact. More details regarding the simulation environment used for this evaluation can be found in Section 4.5.

As depicted in Figure 4.4 the same observations can be extracted as with the real results. More specifically, application overheads change in a linear way and the impact of distance to memory-bound applications is becoming more dominant as the distance of the core to the memory controller increases.

Based on these measurements the distance factor *D(H)* is defined using IPC values at each core distance as described in Equation 4.1. To determine coefficient *a*, *IPCx* and *IPCy* are measured at two different cores, of same core frequency, with distance *Hx* and *Hy* to the memory controller. These measurements can classify an application, based on coefficient *a*, if it is influenced by distance or not as presented in Figure 4.3.

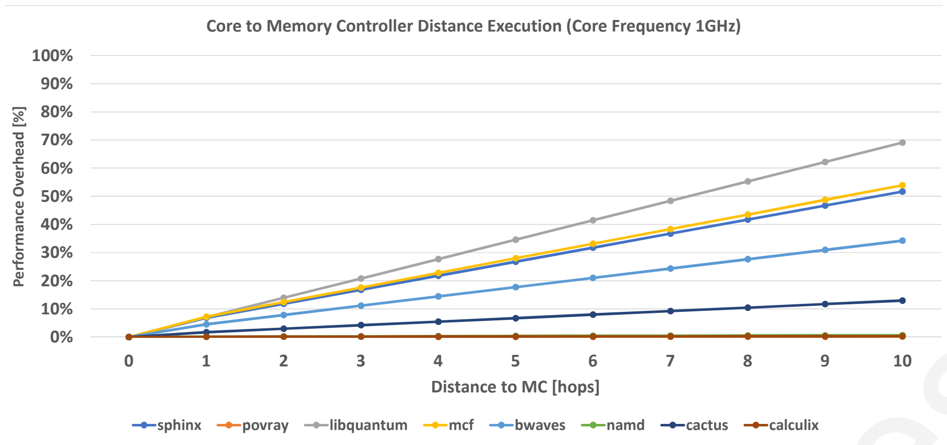


Figure 4.4: Core to Memory Controller Distance Effect Simulated Execution Performance Influence for SPEC applications.

4.3.3 Asymmetric Cores

The second factor studied on a clustered many-core architecture is cores of different computational capabilities and more specifically, different cores frequencies and the impact on the applications' performance. Three different frequencies are selected based on the Intel SCC specifications. More specifically, the default core frequency configuration of 533MHz is selected as the baseline, the 266MHz, as the low-power cores, and 800MHz, which is the maximum available core frequency on the system as the high-performance cores.

Figure 4.5 shows the influence of cores' frequency to applications performance. Results are normalized to the execution on the default 533MHz core frequency and at the same core to memory-controller distance. The first observation is, compute-bound applications, *ep* and *povray*, are influenced the most from core frequency. Memory-bound applications, *dc*, *bt*, *lu*, *sp*, *ua*, *sphinx* and *libquantum*, do not show significant performance degradation from core frequency configuration.

$$\begin{aligned} F(f) &= b \times f, \\ b &= \frac{IPC_x - IPC_z}{f_i - f_j} \end{aligned} \tag{4.2}$$

Equation 4.2 presents the core frequency factor *F(f)* in terms of core frequency *f*. In order to determine coefficient *b* *IPCx* and *IPCy* are measured at two different cores with frequency *fi* and *fj* respectively having the same core to memory controller distance. Having these measurements, the application can be classified if core

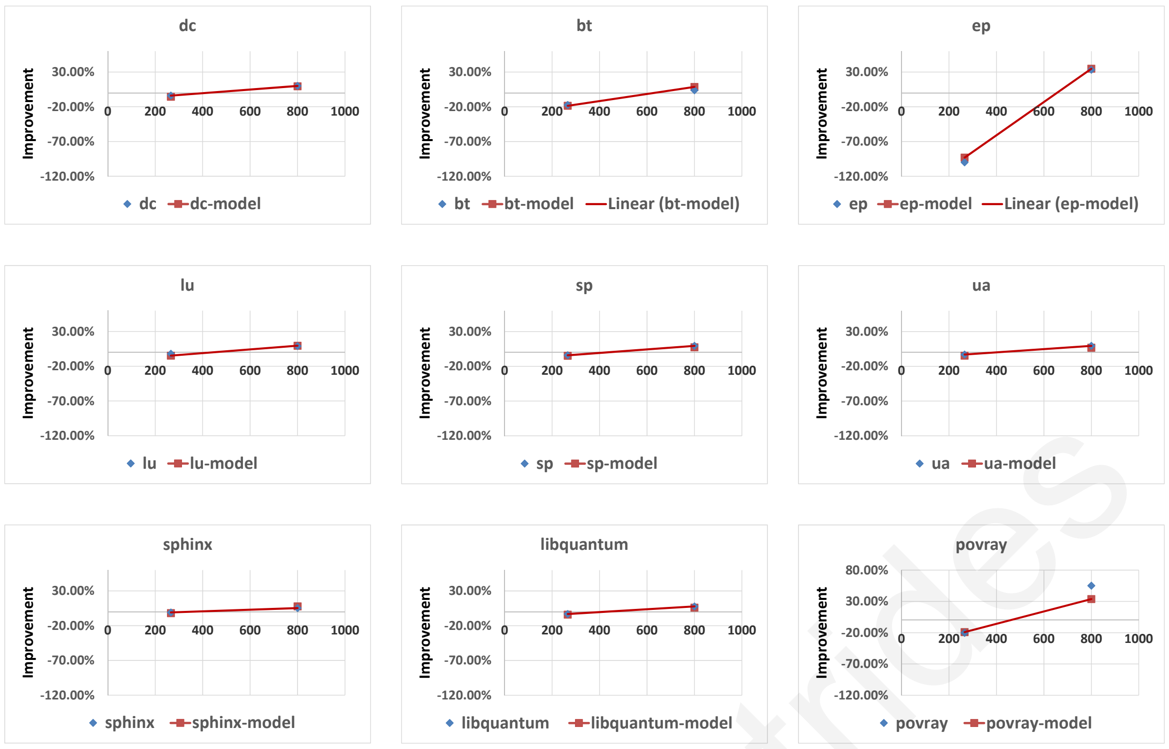


Figure 4.5: Non-Uniform Execution Influence for SPEC and NAS applications.

frequency influences its execution as depicted in Figure 4.5.

In order to capture this behaviour, applications execution is monitored collecting their IPC metrics. Figure 4.5 depicts for each application its behaviour in terms of execution time and the same time its behaviour in terms of IPC measured (model graph). It can be observed that each applications behaviour can be described by monitoring the IPC metric.

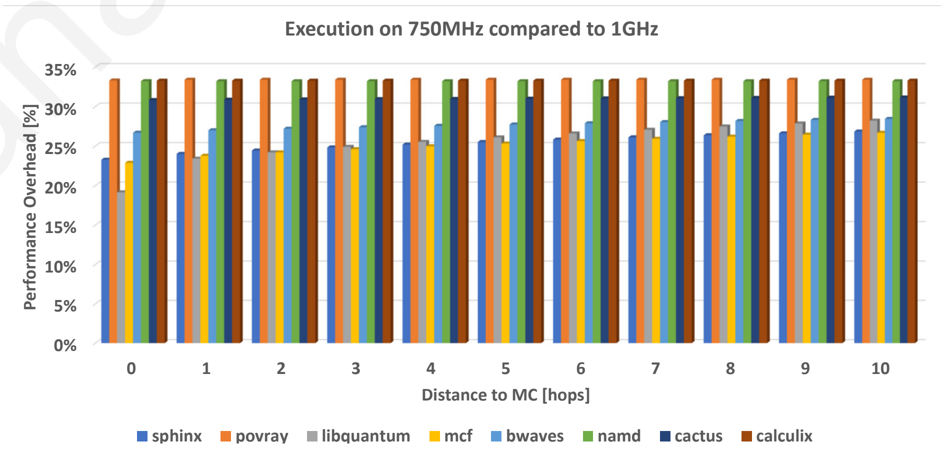


Figure 4.6: Core Frequency and Location Effect for Simulated Execution of SPEC applications.

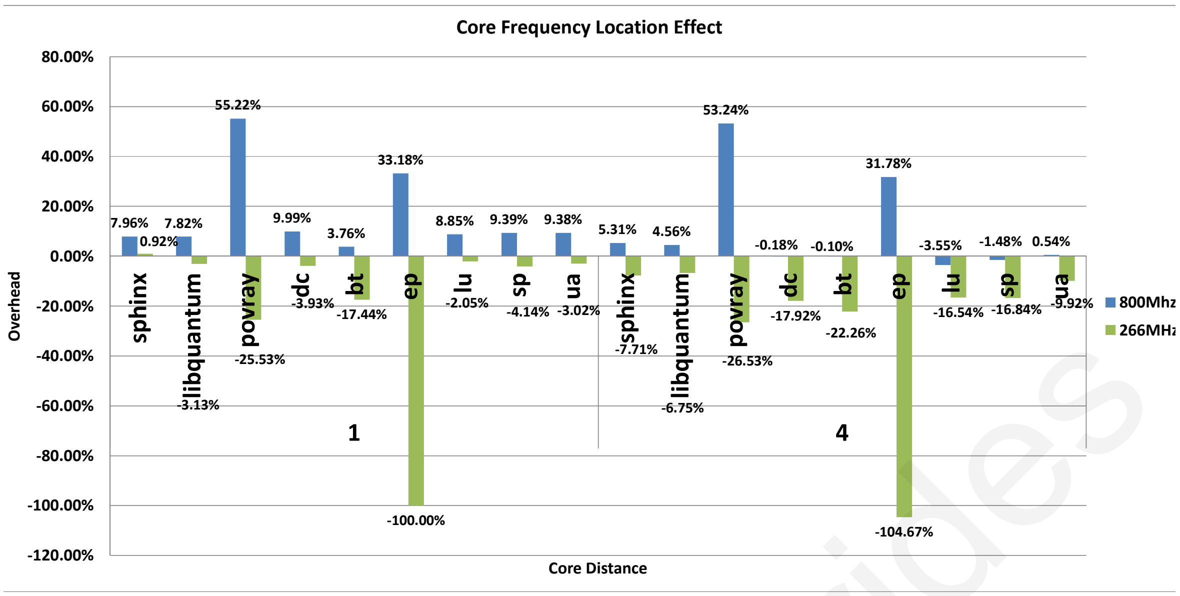


Figure 4.7: Core Frequency and Location Effect

4.3.4 Aggregate Off-chip Bandwidth

In order to study the pressure on the memory controller, due to the combined bandwidth requirements of multiple applications, it is examined how the performance is affected when applications are co-executed on all cores of a cluster. The worst case scenario would be to execute only memory-bound applications in all cores of a cluster. In future large-scale many-core processors the number of cores per cluster will increase faster than the number of on-chip memory controllers making the problem more pronounced. This is due to the fact that the available area and the off-chip pins do not allow the increase in the number of on-chip memory controllers. Moreover, even if it is considered that the number of memory controllers will scale with the number of cores, it is still valid to assume that a memory-aware mechanism will be needed in order to distribute the memory requirements across the different controllers. Therefore, it is important to study the bandwidth utilization on future large-scale many-core processors executing applications' which are either memory- or compute-bound. The thickness of arrows depicted in Figure 4.1 show the cumulative memory bandwidth requirements as processes concurrently demand access to off-chip memory. Figure 4.8 shows how the limited off-chip bandwidth affects the execution time applications when all cores served by a memory controller are occupied. As a baseline it is considered the execution of the corresponding application

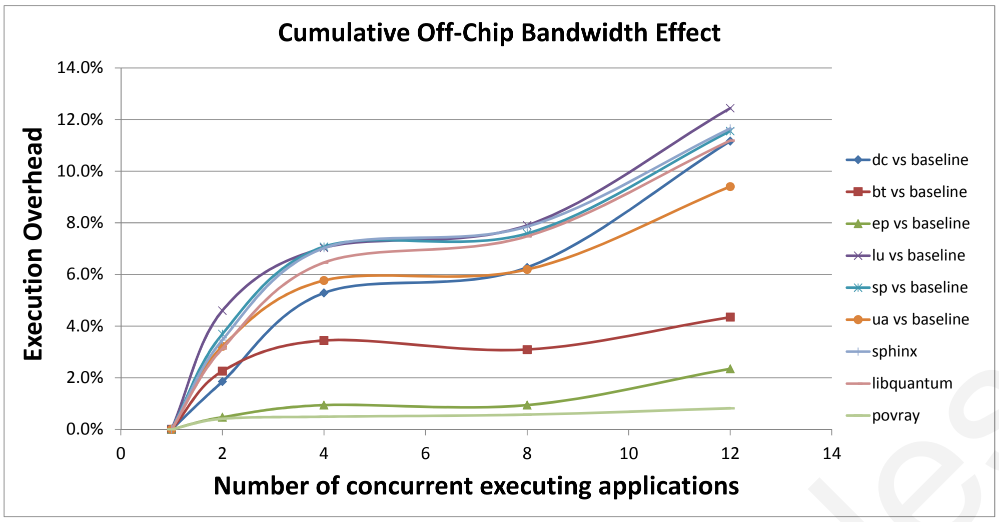


Figure 4.8: Cumulative Bandwidth Performance Influence for SPEC CPU2006 and NAS applications.

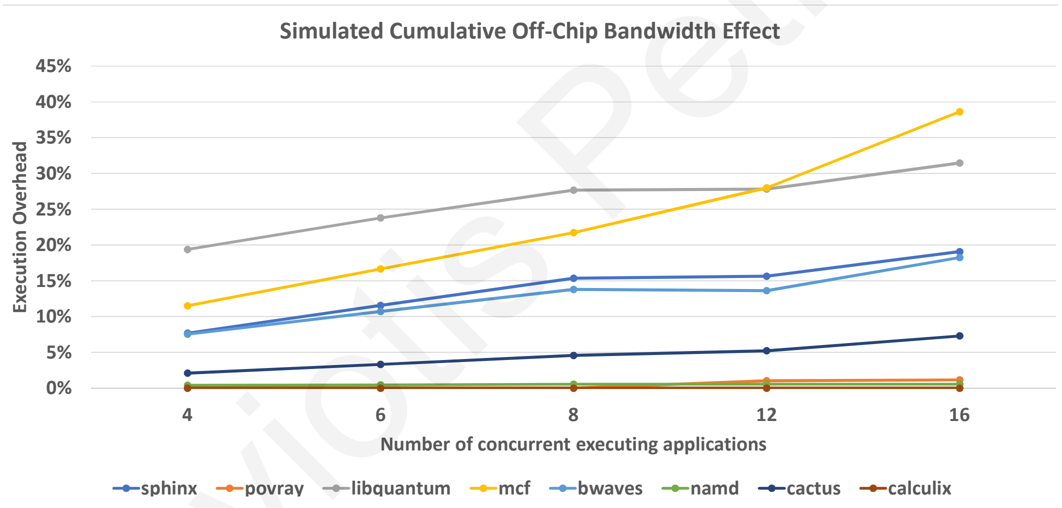


Figure 4.9: Simulated Cumulative Bandwidth Performance Influence for SPEC CPU2006.

alone in a cluster and on the closest core to the memory controller. Results show that when all cores of a cluster are busy a performance overhead of memory-bound applications is observed whereas for compute-bound applications this effect is limited. This influence increases gradually and reaches the maximum when all the cores of a cluster concurrently execute applications resulting to a performance overhead of up to 12%.

4.3.5 Understanding and Classifying Applications Behaviour

Table 4.1 presents the values calculated for both coefficient a and b. From the measurements it is possible to classify each application either as memory or compute-bound by comparing the values of the two coefficient, if $a > b$ then is classified as memory-bound otherwise compute-bound, as depicted in Figure 4.10. It should be mentioned that results are aligned with [41] where applications were classified either as compute- or memory-bound based to their Misses Per Kilo Instructions (MPKI).

In Figure 4.7 it is demonstrated the impact of both cores' frequency and distance to memory controller normalized to the default core frequency of 533MHz with distance 1 (baseline execution). More specifically, it represents the overhead of applications' execution to the corresponding resources, *i.e.* the increase to the execution time compared to the corresponding baseline execution. Results show that cores with low frequency should be placed closer to the memory controller and cores of high frequency should be placed on cores with higher distance in order to minimize the effect of both frequency and distance to application performance. More specifically, it can be observed that applications placed on a low frequency, *i.e.* 266MHz, and high distance to the memory controller, *i.e.* 4 hops to the memory controller, demonstrate large performance degradation. This effect becomes more important on memory-bound demanding applications. Based on the above observations and results the selected architecture configuration is selected as the one shown in Figure 4.2.

As presented earlier both core frequency and distance effect can be described by linear functions, therefore applications behaviour can be predicted by measuring their behaviour on cores where one of the effects factors remains constant and the other one varies in order to isolate the effects. For example, in order to measure distance effect the behaviour of the application is measured on two different distances ensuring cores' frequency constant. Taking the two measurements applications behaviour on different distances can be described. With the same concept, by changing cores' frequency and ensuring constant core-to-memory controller distance, applications behaviour for different cores frequencies can be described. As presented previously the IPC for each application is monitored varying only one factor. This approach was selected since the architecture of the Intel SCC does not offer any performance counters to monitor the extended L2 cache memory.

This proposal can predict applications behaviour on different resources if the previously mentioned conditions are satisfied. More specifically, if pairs of resources with one effect variable and the other one constant exist on the system, it is possible to determine the effect of each factor on applications execution. In addition, it

Table 4.1: Applications classification memory or compute-bound based on coefficients a and b.

Application	*Factor a*	*Factor b*	*Classification*
dc	0.030	0.003	Memory bound
bt	0.014	0.004	Memory bound
lu	0.047	0.002	Memory bound
sp	0.049	0.003	Memory bound
ua	0.044	0.002	Memory bound
sphinx	0.032	0.001	Memory bound
libquantum	0.034	0.002	Memory bound
mcf	0.040	0.024	Memory bound
bwaves	0.110	0.068	Memory bound
ep	0.013	0.025	Compute bound
povray	0.004	0.014	Compute bound
namd	0.030	0.114	Compute bound
cactus	0.040	0.074	Compute bound
calculix	0.050	0.144	Compute bound

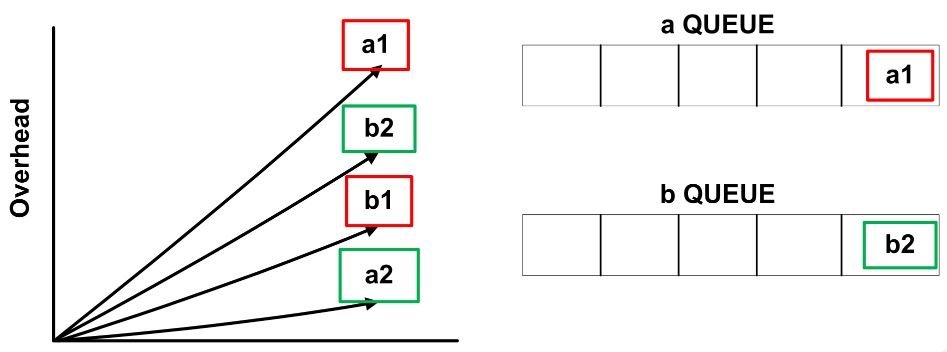


Figure 4.10: Applications Classification comparing coefficients a and b.

must be noted that the classification phase, determining coefficients a and b, can be completed in two steps. As depicted in Figure 4.12, there is a need of two distinct couples of resources where the measurements and conditions can take place.

4.4 Scheduling Policy

In this Section, the proposed scheduling policy and implementation details of the selected case study is presented.

4.4.1 Classification Phase

As described earlier, in order to quantify core to memory controller distance and cores' frequency effect, it is needed to capture their behaviour to the different resources. To achieve this, a classification phase is needed during which coefficients a and b, as presented in 4.3.2 and 4.3.3, are determined for each application.

This work proposes that this phase is performed at the beginning of execution of each application. More specifically, when an application starts its execution on the system it is placed randomly at any available resource. The following metrics are gathered: its IPC, cores' frequency and distance to the memory controller for a specific time slice. On the next time slice this application is moved to another core with the same frequency but different core distance, gathering the same metrics as depicted in Step 1 of Figure 4.12. From the gathered measurements coefficient a of Equation 4.1 can be evaluated and therefore cores' distance effect. At the next time slice applications are moved to another core with different frequency but to an already examined distance. The same metrics are gathered: IPC, cores' frequency and distance metrics as depicted in Step 2 of Figure 4.12. Having these measurements coefficient b of Equation 4.2 can be evaluated and therefore determine cores' frequency effect. It must be mentioned that all the above exchanges are performed within the same cluster of cores in order to avoid any interference of the cumulative off-chip bandwidth to the measurements.

If discrete couples of cores exist, with the above mentioned characteristics, on the system, exchanges of applications to resources can occur as a single migration at each step. Moreover, due to the fact that the classification is performed on-line where other applications are co-executing on the system, migrations of applications that were perfectly assigned may occur. The trade-off of this occurrence is that first knowledge of the application behaviour is earned and secondly that a miss-placed application will be fixed during scheduling phase. It is important to mention that there is no knowledge of applications' behaviour before execution but instead applications are classified during runtime. The classification process continues during the whole execution of applications, since they might enter a different phase of execution with different requirements. In addition, application migration from one core to another has limited overhead impact on their execution and based on the experimental results is less than 1% on the total execution time of the application.

Having coefficients a and b for each application it can be evaluated for each factor how they influence applications' performance. Two discrete queues are constructed, one having applications where $a > b$ and another with the rest of the applications. Consequently there are applications that distance affects their execution and applications which are influenced by cores' frequency. The queues are ordered from largest to smallest value of a and b respectively.

4.4.2 Applications Scheduling

Having the queues created during the classification phase (Figure 4.10), scheduling to system cores takes place. An on-line hierarchical heuristic policy is used of assigning applications to cores. The first criteria is to satisfy applications belonging in a queue by assigning them to cores with the nearest distance. The second criteria is to satisfy applications in the b queue, by assigning them to cores with high frequency. It should be noted that the two queues are exclusive, therefore an application can be included only to one of them.

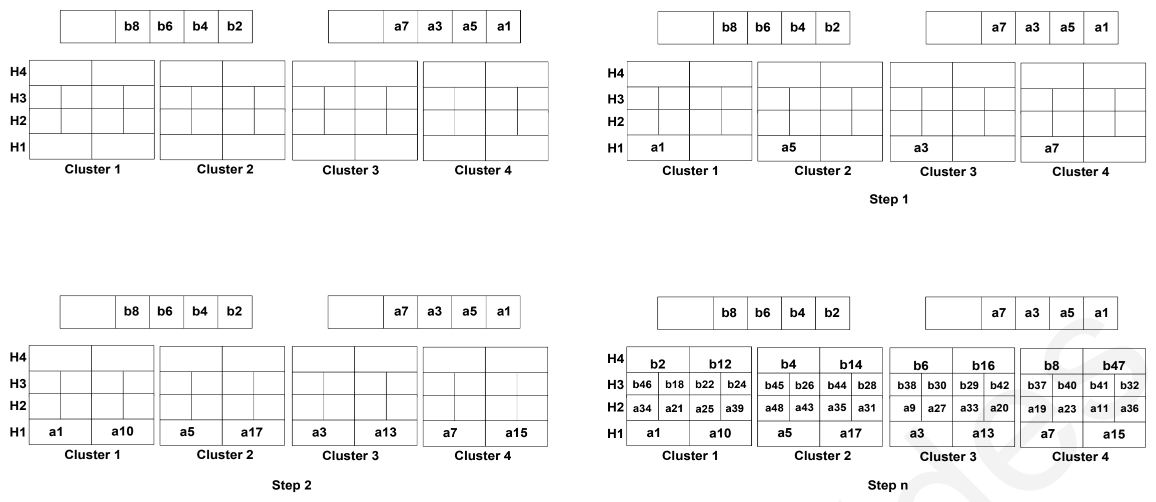


Figure 4.11: Applications to Resources Assignment.

Figure 4.11 shows the steps of the scheduler. First applications that belong to *a* queue are assigned to the nearest cores to the memory controller of the system. More specifically, a round-robin technique assigns applications to different clusters in order, first to exploit all minimum distances and second to optimize utilization of off-chip bandwidth between clusters (Figure 4.11, Step 2). After completing assigning applications of *a* queue, applications included in the *b* queue are assigned (Figure 4.11, Step n). With the same concept using a round-robin technique, applications are assigned to each cluster core whose frequency satisfies applications' behaviour. It should be noted that during this placement low frequency cores are selected only if no other resources of higher frequency are available. Application migration from one core of the system to another adds a limited overhead to their execution, as described earlier is measured to be less than 1% to the total execution time of each application.

In the case where an application is placed to a resource which does not fulfill its criteria, due to the fact that there is no such resource available, it is moved to a better matching resource when it becomes available since it remains at the respective queue.

During the execution of the whole application, it may enter different phases with different requirements. More specifically, for a phase of execution an application becomes more sensitive to distance rather than cores' frequency and vice-versa. In order to capture this behaviour monitoring applications execution continues and compare its measurements. If changes to the metrics are observed then the classi-

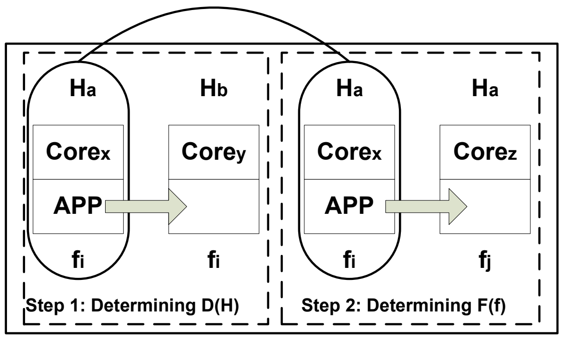


Figure 4.12: Classification Steps for Distance and Frequency.

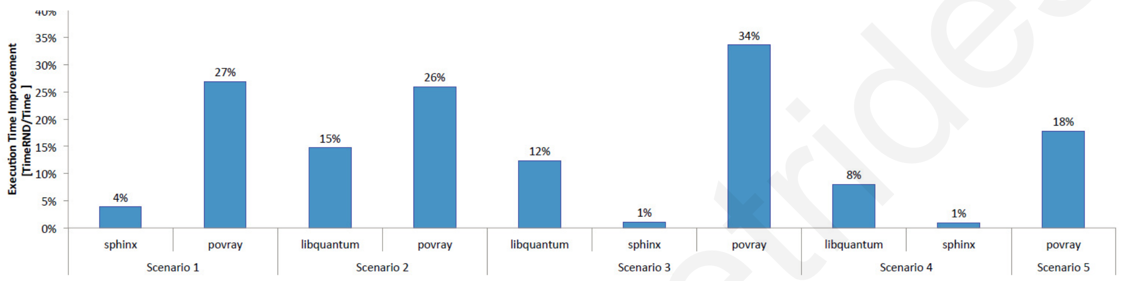


Figure 4.13: Proposed Scheduler Execution Time Improvement compared to a Random Static Task Assignment Policy.

fication phase is performed for the application, within the cluster that is assigned, in order to redetermine the factors that application becomes sensitive. In the case where an application changes its behaviour *a* and *b* are reconstructed by triggering the classification process as described earlier.

4.4.3 Implementation Details

For the purpose of this work real applications are selected from the SPEC CPU2006 benchmark suite [74]. The classification methodology is validated with [75] and [41]. Results show that the proposed classification is aligned with these studies and the classification methodology captures applications requirements.

As it is assumed no previous knowledge about the characteristics of the applications, in order to determine dynamically at runtime the category where each application belongs, their execution is monitored. IPC metrics are collected (using program counters) of the applications and classify their behaviour as, compute- or memory-bound. Information regarding cores status (if it is idle or busy) is collected during runtime. Cores that have the same frequency belong to the same group, which is called from now on frequency domain, and as described earlier in this work, three different frequency domains are considered.

In order to capture applications behaviour during their execution a mechanism is needed that can capture specific metrics such as the memory usage per application at each core, each cores' frequency and state, *i.e.* idle or busy. Applications' executing at each core are characterized either as memory- or compute-bound processes according to the metrics gathered. Memory-bound processes are stored in the *a* queue and compute-bound processes are stored in the *b* queue. These queues are updated and reconstructed if a change to applications behaviour is identified. Finally, these queues are stored on the host PC where the scheduling policy is executed.

The queues created are used by the scheduler which is responsible for mapping applications to system cores that will provide best match. The scheduling policy aims to satisfy the requirements of applications executing on the system considering not only the characteristics of processes but also the system characteristics as described earlier.

Due to the targeted architecture, where the experiments and scheduling policy were tested, specific details had to be addressed. More specifically, the Intel SCC experimental processor is a clustered many-core architecture and therefore process migration between cores is not as easy as in shared memory architectures. In order to address this issue a well known technique used on clustered processor architectures is adopted. This technique is known as process check-pointing and its purpose is to save the state of a process executing and resuming its execution from that same point. To achieve this, the cryopid [76] check-pointing library was ported to the Intel SCC research processor. Cryopid can be linked to existing applications and provide independent check-pointing functionality without the need of any kernel modifications. As the library is part of the application's process, it can access all resources in the same way the original application does. Moreover, it can take checkpoints at any given time, being independent of time-frame constraints imposed by the application. This technique is effective and of low overhead and therefore can be adopted in order to achieve this work target. More specifically, the overhead of check-pointing and restarting the application from one core to another is up to 1% to the total execution time of the corresponding application executing on a core clocked on 533MHz and close to the memory controller.

Tasks migration from one core to another is triggered by the Task Assignment Policy, which determines the cores that will participate. It is considered that a future large-scale many-core architecture will have the characteristics of a clustered many-core architecture and therefore it should be performed in two steps. First, it checkpoints and freezes the execution of processes currently executing on the cores by sending a signal to the corresponding core. Secondly, it triggers the resume of processes execution on the destination cores as determined by the scheduler.

4.5 Experimental Setup

Table 4.2: SPEC CPU2006 Applications Scenarios and Execution Times Variance.

Scenario	*Applications*	*Execution Time RND (sec)*			*Execution Time Scheduler (sec)*
		Worst	*Best*	*Average*	
1	povray	6885.6	4431.1	6266.7	4581.5
	sphinx	45986.2	35015.8	39584.7	38068.2
2	povray	6885.65	4432.1	6265.8	4640.8
	libquantum	55007.1	45370.4	48473.6	41327.8
3	povray	7486.9	6436.1	6961.5	4621.4
	sphinx	39407.5	38129.2	38768.4	38370.5
	libquantum	50301.9	44595.2	47448.6	47448.6
4	sphinx	41919.6	33501.6	38710.5	38479.3
	libquantum	55167.1	47375.9	49645.2	45502.8
5	povray	10936	3126.2	6640.3	5460.6

For the experiments of this work applications from the SPEC CPU2006 [74] and NAS [77] benchmark suite are used. The applications were selected according to their characteristics, as mentioned in Section 4.3, using their reference input data sets. Applications of memory- and compute-bound characteristics are selected. For this work the Intel SCC experimental processor, RockyLake version, is used as real clustered many-core architecture. The system main memory was configured with 32GB in total. By changing the voltage/frequency domains of the Intel SCC experimental processor [24] three different domains are created, each composed of 16 cores. The maximum available frequency configuration of a core on the Intel SCC research processor is 800MHz and the default is 533MHz. These frequencies are selected as the high-performance frequencies. In addition, 266MHz core frequency is selected as the low-power core frequency. The 48 cores are split into three voltage frequency domains (266MHz, 533MHz, and 800MHz) of 12 cores, as depicted in Figure 4.2, in order to emulate the heterogeneity of the system. The mesh interconnection network, the DDR3 memory and the Memory Controllers were all clocked at the default frequency of 800MHz. The Operating System used for the Intel SCC cores is the default Linux kernel provided by the RCCE SCC Kit 1.4.0. The application checkpointing and resume is performed using the Cryopid [76] checkpointing library which was ported to the Intel SCC experimental processor. The power consumption is measured using the same technique used by the SCC GUI performance meter by reading the FPGA emulated registers that hold the appropriate values. The random static assignment is selected as the baseline for comparing the results of the proposed scheduler.

Table 4.3: Clustered Many-Core Architectures Environments.

Configuration	*Intel SCC*	*Simulation*
Off-chip Bandwidth per Memory Controller	1.6 GB/sec	32 GB/sec
Cores per Memory Controller	12	32
Core Architecture	IA-32 *x86* (P54C)	Nehalem *x86*
Cores Frequency	800 MHz 533 MHz 266 MHz	1 GHz 750 MHz
L1 Instruction cache size	16KB	32KB
L1 Data cache size	16KB	32KB
L2 cache size	256KB	512KB

4.6 Experimental Results on the Intel SCC

In this work analysis the proposed scheduler is evaluated according to two different metrics: *(i)* performance and *(ii)* energy-delay product (EDP). The presented analysis compares the results gathered from the proposed scheduler with a random static assignment policy (RND). The scenarios selected for the evaluation of the proposed scheduler are depicted in Table 4.2. Random static assignment was selected as the baseline according to which processes are statically executed on the preassigned core. The results presented are the average obtained from the execution of 5 different random assignments for each scenario. According to the number of applications and the possible scenarios, only the most representative ones have been selected. For each scenario it is assumed that the applications are initially randomly assigned to the system cores and all cores of the system are occupied [75]. The proposed scheduling policy was evaluated on how it performs in the cases where: *(i)* equal number of compute- and memory-bound are executing (Scenario 1 and 2); *(ii)* different memory- and compute-bound applications are co-executing (Scenario 3); *(iii)* different memory-bound applications executing (Scenario 4); and *(iv)* only compute-bound applications are executing (Scenario 5).

The first evaluation of the proposed scheduler shows its impact on the execution time of the selected applications co-executing on the target architecture. Table 4.2 depicts the execution time for the random static assignment and the proposed scheduling policy. It is important to notice that random static assignment suffers from variance. This is a result of two factors: *(i)* applications can be assigned to a core that does not satisfy their requirements and *(ii)* the core to memory controller distance and the off-chip bandwidth for memory-bound applications is not considered. The proposed scheduling policy results are below the average of the random static assignment for almost all scenarios. In addition, they are considerably lower than the worst random execution cases. In Figure 4.13 it is presented the average improvement of execution time for the proposed scheduling policy over the random static assignment for the different scenarios. From the results a significant performance improvement in both memory- and compute-bound applications can be observed. More specifically, *povray* shows a performance improvement of up to 36%, in Scenario 1. In addition, memory-bound applications achieve a performance improvement up to 15%, in Scenario 2. Finally, it is important to mention that an overall performance improvement of all applications in the different scenarios is observed. It is important to notice that the results presented in Figure 4.13 include the cost of migration and thus it can be concluded that there is low overhead to the execution times of the applications due to migration.

Lets consider the scenario in which equal number of memory and compute-

Table 4.4: Moving memory-bound applications closer to the memory controller, Scenario 1.

Initial Core (Distance)	*Application*	*Destination (Distance)*	*Improvement*
Core 24 (1)	sphinx	-	4.39%
Core 25 (1)	sphinx	-	3.71%
Core 26 (2)	sphinx	Core 00 (1)	2.27%
Core 27 (2)	sphinx	Core 01 (1)	2.26%
Core 36 (2)	sphinx	-	4.23%
Core 37 (2)	sphinx	-	3.82%
Core 38 (3)	sphinx	Core 12 (2)	3.59%
Core 39 (3)	sphinx	Core 13 (2)	2.59%

bound applications are co-executing in the same voltage/frequency domain (*sphinx* and *povray* in the 533MHz domain), Scenario 1. In this scenario memory controllers are initially responsible for only one type of application. In this case, the intra-domain application migration phase takes place exchanging an equal number of compute- and memory-bound applications from one controller to another bringing memory-bound applications closer to the memory controller. Table 4.4 depicts the results of exchanges between applications on cores closer to the memory controller on Scenario 1. Both initial and destination cores belong to the same voltage/frequency domain but they are connected on different memory controllers. One important observation of these results is that in this case memory-bound applications can benefit by utilizing the memory bandwidth and bringing them closer to the memory controller. It is important to notice that even though applications in cores 26, 27, 38 and 39 migrate from one core to another, overall they show improvement in their performance. From the experimental results it is shown that the proposed scheduling policy improves applications performance by satisfying their requirements during their execution.

Figure 4.14 presents the energy-delay product (EDP) of the proposed scheduler compared to RND. As energy-delay product is considered to be the product of energy consumption and the total time of execution of the application. For example, for the proposed scheduling policy EDP is calculated as the sum of the products of the

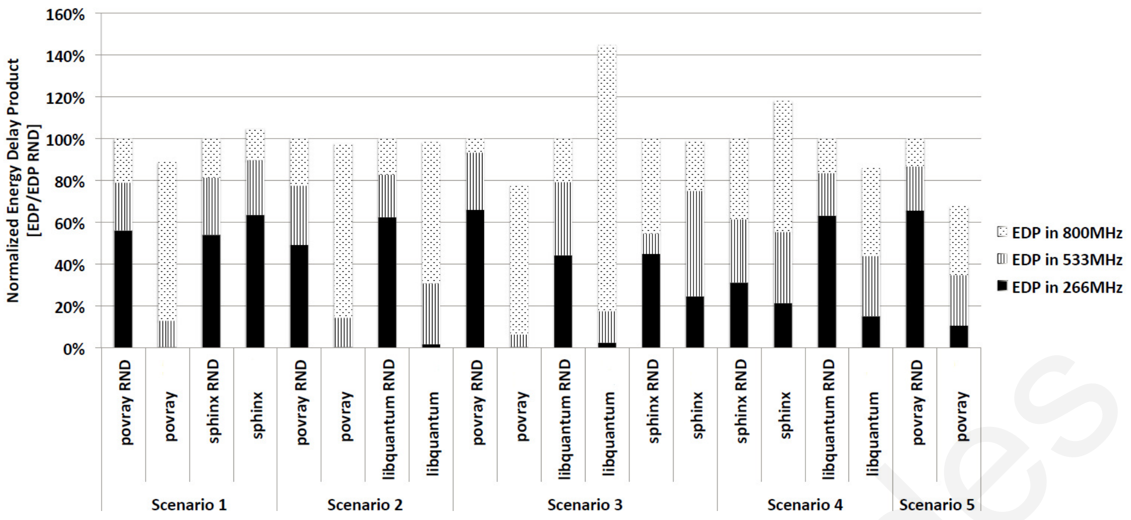


Figure 4.14: Energy Delay Product Comparison between a Random Static Assignment Policy and the Proposed Dynamic Scheduling Policy.

energy consumption at each voltage/frequency domain, in which the application is executed, multiplied by the time that the application executes at each domain. The results are normalized to the EDP result of each application using the static random assignment on the 266MHz domain. Each bar shows the contribution of each domain through the whole execution of each application. The first conclusion is that the proposed scheduler benefits the compute-bound applications in terms of both performance and power consumption on an average of 15% for all executed scenarios. This is because compute-bound applications have a priority over memory-bound applications for the execution on domains with higher voltage/frequency and therefore their execution time is reduced. Studying these results more carefully it can be observed that memory-bound applications show an increase on the power consumption of the system. This behaviour arises from the fact that memory-bound applications are assigned to domains of lower voltage/frequency and are only assigned to domain of higher voltage/frequency whenever a core of such a domain is available. Even though someone could expect that this would have a significant impact on their performance, results show that this performance benefit comes at a power cost. Finally, in the case of Scenario 3 it is possible to observe that after the completion of the compute-bound application, the memory-bound applications compete between themselves for the high voltage/frequency domain cores. This results in the high overhead in the efficiency for 42% as observed. From the results presented it is important to observe that the proposed scheduling policy is a greedy policy and thus benefits compute-bound applications in terms of both performance and power consumption. As for the memory-bound applications it is possible to observe that while their performance is improved, in some cases there is a penalty regarding their power consumption.

4.7 Simulating Clustered Many-core Architectures

In order to study the effectiveness of the proposed scheduling policy on future large scale many-core architectures, it is necessary to use a simulator to test different design points from the Intel SCC. More specifically, the number of cores per cluster is scaled to 32. In addition, each cluster consists of more sophisticated cores, *i.e.* Nehalem cores, and each memory controller dedicated to a cluster has a maximum off-chip memory bandwidth of 32GB/sec as depicted in Table 4.3. Within the cluster equal number of cores of the selected core frequencies, 750MHz and 1GHz, co-exist. Finally, equal number of applications are co-executing within the cluster for each scenario.

The described configuration of the simulated clustered many-core architecture was selected for the following reasons: *(i)* to investigate if the proposed scheduling policy is effective as the number of cores within the cluster increases; *(ii)* to further evaluate the proposed scheduling policy as the diversity of cores within a cluster changes, forming other possible configurations of clustered many-core architectures.

4.7.1 Scaling the Number of Cores within a Cluster

As described previously, the effectiveness of the proposed scheduling policy is examined when the number of cores within a cluster scales to 32. First the potential performance of each application executing in each scenario is evaluated. More specifically, for each scenario it is measured the worst and best execution of each application while co-executing within the system. Figure 4.15 depicts applications execution between the best and the worst placement of an application to the cluster cores. For example, in Scenario 1 if *povray* has a potential performance benefit of about 17% when it is placed to a best matching resource compared to a non matching resource. From the presented results, it can be observed that it is important for applications to be placed to a best matching resource since satisfying their requirements

Table 4.5: Simulated SPEC CPU2006 Applications Scenarios.

Scenario	*Applications*	*Scenario*	*Applications*
1	povray sphinx	6	bwaves libquantum
2	povray libquantum	7	cactus bwaves
3	povray sphinx libquantum	8	cactus sphinx
4	libquantum sphinx	9	cactus libquantum
5	calculix bwaves	10	cactus namd

in terms of resources can result in overall performance improvement. It must be noted that in some cases a limited potential performance degradation is observed. This is a result of having applications of the same requirements in terms of resources co-executing within the system, therefore competing for the same resources. It is considered that the best potential performance in such a case to be the one with limited impact to the less demanding application and maximizing the performance of the most demanding application. For example, in Scenario 4 both applications *sphinx* and *libquantum* are memory-bound but *libquantum* is more demanding than *sphinx* (*factor a > factor b*).

To evaluate the proposed scheduling policy the same scenarios are executed for the simulated clustered many-core architecture and the proposed scheduling policy is enabled. In Figure 4.16 the results of each application for each scenario are depicted. More specifically, results show how close to its potential execution each application is (as defined previously in Figure 4.15) when the proposed scheduling policy is executed. From these results it can be observed that almost for all scenarios the proposed scheduling policy manages to achieve the potential performance of each application. Moreover, it is important to note that the proposed scheduling policy can mitigate well the increasing number of cores within a cluster and maintain

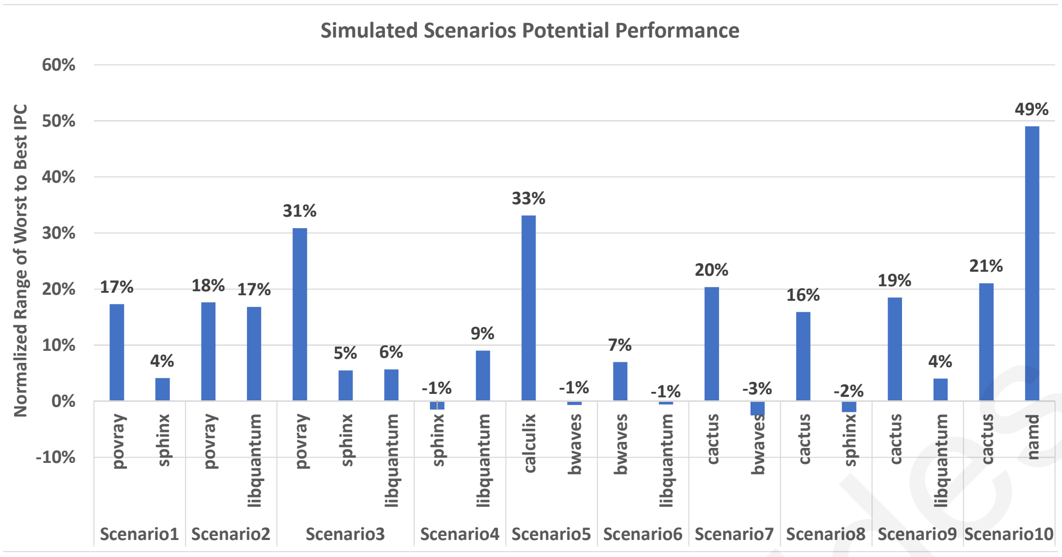


Figure 4.15: Simulated Scenarios Potential Performance.

its effectiveness to applications performance.

Furthermore, the effectiveness of combining both core frequency and core distance factor is studied. In Figure 4.17 the achieved performance compared to the potential for each scenario application is presented. More specifically, the proposed scheduling policy is executed by enabling only one factor (*single factor*), either core distance (*Distance*) or core frequency (*Frequency*). The results depicted in Figure 4.17 show that considering both factors to the scheduler decisions can result on higher overall applications performance. Moreover, it is important to note that the proposed scheduling policy does not show high variances through the different scenarios compared to a single factor scheduling policy. Nevertheless, the proposed scheduling policy shows a consistency through the different scenarios achieving high potential performance of executed applications.

4.7.2 Changing Cores Diversity within a Cluster

Additionally to the previous experiments, the proposed scheduling policy behaviour as the diversity of cores within a cluster changes in terms of number of cores is studied. More specifically, its behaviour is studied when the ratio of low end cores (clocked at 750MHz) are occupying the 75% of the total number of cores of the cluster. In Figure 4.18 is depicted both single factor scheduling and the proposed scheduling policy results for selected scenarios. Results show that the effectiveness of the proposed scheduling policy remains at high levels and at the same time outperforms

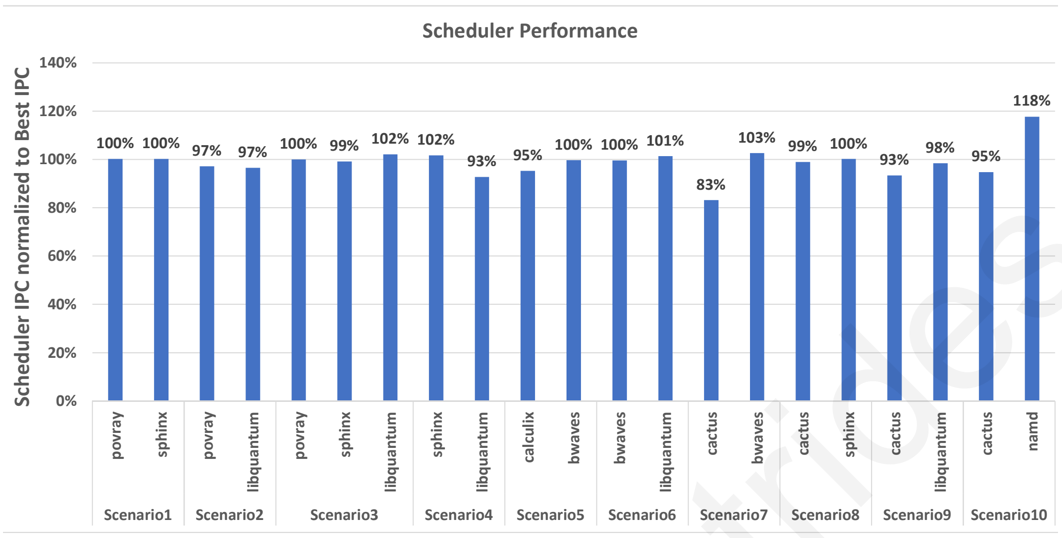


Figure 4.16: Simulated Scheduler Scenarios Achieved Performance.

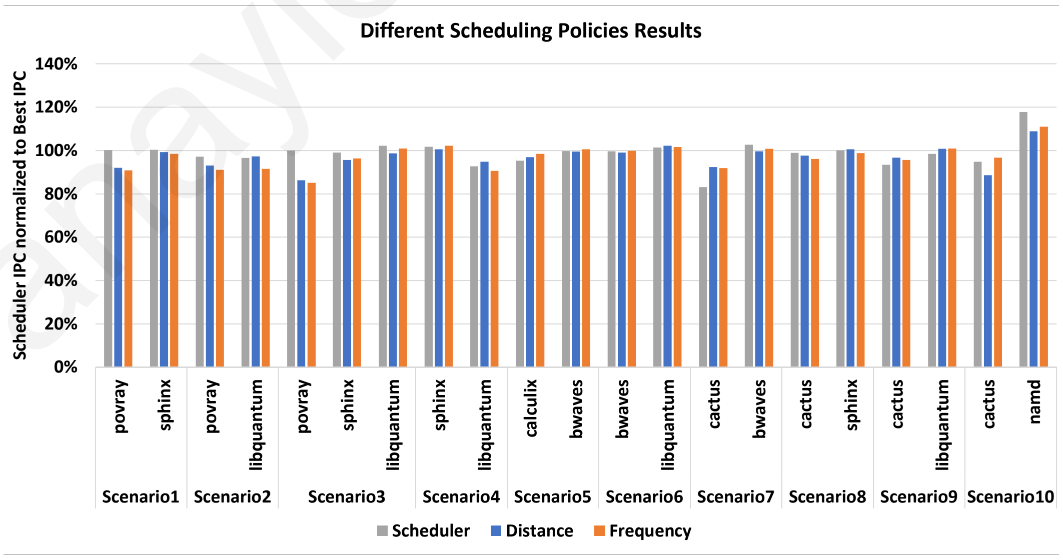


Figure 4.17: Different Policies Performance.

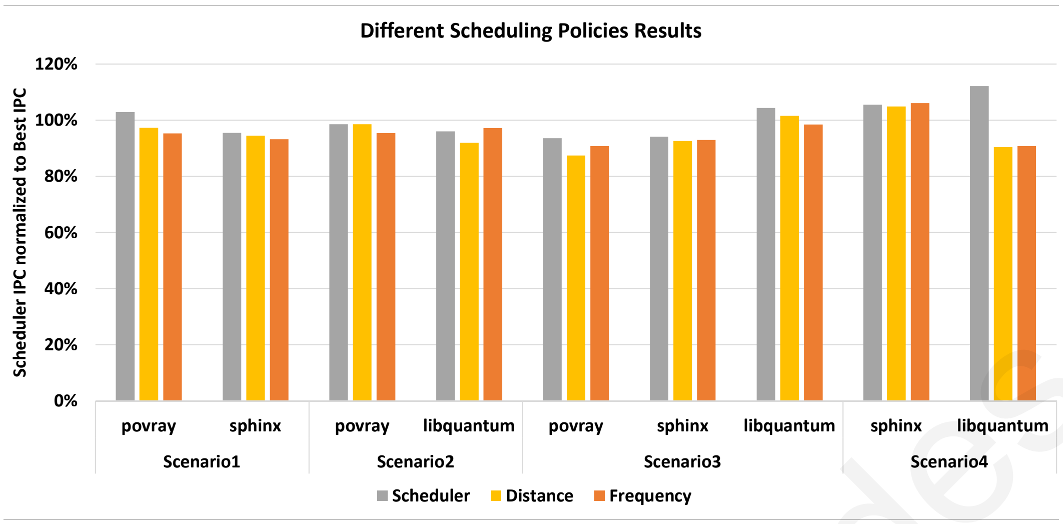


Figure 4.18: Different Policies Performance.

the single factor scheduling. Moreover, the proposed scheduling policy maintains a stable performance through the different scenarios compared to the single factor scheduling which shows a high variance.

4.8 Discussion

Future many-core architectures will come with many challenges in order to exploit their potential performance. This work identified specific challenges of such architectures. More specifically, the challenges identified and which influence applications performance are: *(i)* the core to memory controller distance effect, *(ii)* cores diversity in terms of cores' frequency and *(iii)* the off-chip memory controller contention. The proposed solution of addressing these challenges is a portable online NUMA-aware heterogeneous scheduling policy which tries to satisfy applications characteristics and demands according to the available resources. Moreover, the factors used in the proposed scheduling policy can be used either combined or stand alone (single factor scheduling) in order to be used according to the target system characteristics. For example, if a system is identified that only cores' frequency is influencing application performance then it could use the single factor scheduling to satisfy applications demands to the available resources. Finally, the scheduling policy is implemented in a modular way using the different factors for identifying applications characteristics. Therefore, other factors may be added to capture other characteristics or behaviour of executed applications within a system and thus extend its applicability to other architectures as well.

The portability of the proposed online NUMA-aware heterogeneous scheduling policy it is shown by porting it to a simulator to study other many-core architectures. The experimental results show that the befits are not limited to the Intel SCC processor, but it could mitigate and perform as designed to other architectures as well. Moreover, it can be considered that the proposed solution can be ported to other existing many-core architectures, such the Intel Xeon Phi multi-core [78], which can be seen as the commercial successor of the Intel SCC processor [79]. Therefore, the proposed scheduling policy could also be ported to the Intel Xeon Phi multi-core addressing the same challenges and exhibit the performance potentials as exploited for the Intel SCC processor.

Furthermore, as described earlier, a system may have resources that may affect applications execution by a single factor, *i.e.* cores' characteristics. An example of an architecture that includes cores of different characteristics is the big.LITTLE by ARM [80]. In this case, the proposed scheduling policy could be used with the single factor scheduling implementation and considering only the cores' capabilities and characteristics for satisfying applications demands.

From the above presented examples, it is depicted that the proposed scheduling policy it is not limited to a specific architecture. On the contrary, it can be ported and maintain its applicability to existing and widely used architectures of different characteristics.

4.9 Summary

In this chapter a dynamic scheduling policy is proposed that tries to address the factors that affect the performance of applications executing on future heterogeneous NUMA many-core processors. This work is implemented both on a representative many-core architecture, the 48-core Intel SCC processor, and on an alternative simulated many-core architectures. This work has shown via experimental results that these factors are: (i) non-uniform memory latency, (ii) the different characteristics of cores in such architectures and (iii) the limited off-chip bandwidth offered by the memory controller to the cores. The experimental results for the proposed scheduler show that satisfying applications requirements during their execution can improve significantly their performance. In particular, compute-bound applications can improve their execution time up to 36% and memory-bound applications up to 15%. Moreover, experimental results show that the energy efficiency of the system is improved, by approximately 40%, and the same time achieve a very good performance, by approximately 18%, for compute-bound applications. Given the priority of the proposed policy, memory-bound applications show smaller improvements in both performance and power consumption. The results are very encouraging for the use of such feature-aware assignment policies for future many-core processors.

Chapter 5

Modular Virtualization Layer

In the recent years there has been a shift in processor architecture towards chips with multiple cores, thus avoiding the power and complexity walls. The increasing number of cores will lead in the future to three major challenges: (1) core memory hierarchies configuration, (2) management of such complex hardware, and (3) programmability and/or portability for such systems. In order to achieve a better match between the hardware and the demands of different applications and their phases, future processors will have to offer cores with different specifications which could even change dynamically at run-time. For addressing these issues it is possible to envision that for future processors the hardware will be packaged along with a virtualization layer that hides the hardware complexity and at the same time monitors the application behavior as to transparently improve its performance at run-time. The virtualization layer must be built in a modular way including a core component and providing to the Operating System (OS) and programmer a standard interface to utilize the hardware.

5.1 Motivation

Multi-core processors have been introduced as a solution to continue the performance increase rate and at the same time keeping the design within the required power budget. Increasing the number of devices on a chip not only will offer the benefit of increasing the potential for parallelism but also it will allow manufacturers to explore new designs such as including in the same chip cores of different characteristics. As different applications, and even different phases of the same application, have different demands, a processor with a diversity of cores would be able to achieve a better application-to-hardware match [81]. Consequently, this results in a better power-performance efficiency.

Clearly, this increasing of the number of cores on a chip along with the fact that diverse types of cores will be available, requires a higher management effort. Currently it is already possible to observe this issue as it is difficult to efficiently port a certain application for different available multi-core architectures [82]. It is therefore only natural to expect that the effort involved in tuning applications will increase dramatically for larger scale multi-core chips that include cores of different characteristics, which in some cases could even change their configuration dynamically at run-time. This work proposes that future multi-core architectures contain processing cores and memory hierarchies that are able to change their configurations at run-time. These can be called Morphable multi-cores.

This effort is currently exported up to the level of the programmer who has in many cases to use different languages and libraries in order to exploit the benefits of different architectures [83, 84]. Some effort is currently underway in expressing the parallelism using abstractions as to be able to generate the code automatically for different platforms like Intel Parallel Studio [85], Rapidmind [17], Intel's Ct [86], and OpenCL [87]. Nevertheless, in most cases the use of general constructs results in significant performance overheads [11] as it is hard to efficiently map programs to the different architectures.

As technology advances and architectures change, tuning the same applications over and over for the new architectures becomes an overwhelming task. Also, by using the same core designs, manufacturers are able to produce many different processors, depending on the number of available cores and their configuration. The objective of this work is a virtualization layer or hypervisor which will hide the complexity and diversity of the hardware as depicted in Figure 5.1. This virtualization layer operates as the manager of the underlying Morphable multi-core, releasing the programmer from this demanding task, and also hiding some complexity of the system from the OS. In other words, by offering this virtualization layer along with the hardware it is possible to offer a standard set of core services to the upper layers, such as thread scheduling, memory prefetching, and hardware reconfiguration. For example a regular OS could use the scheduling services provided by the virtualization layer to do the mapping of the tasks among the available cores. This mapping could be as simple as just randomly distributing the threads among the different cores or as complex as making architecture-aware decisions that, based on online monitoring information of the application behavior, are able to select the best matching cores available. The mentioned services are supported by a group of mechanisms transparent to the user/OS. Examples of such mechanisms are: resource detection, thread monitoring, fault-tolerance, among others.

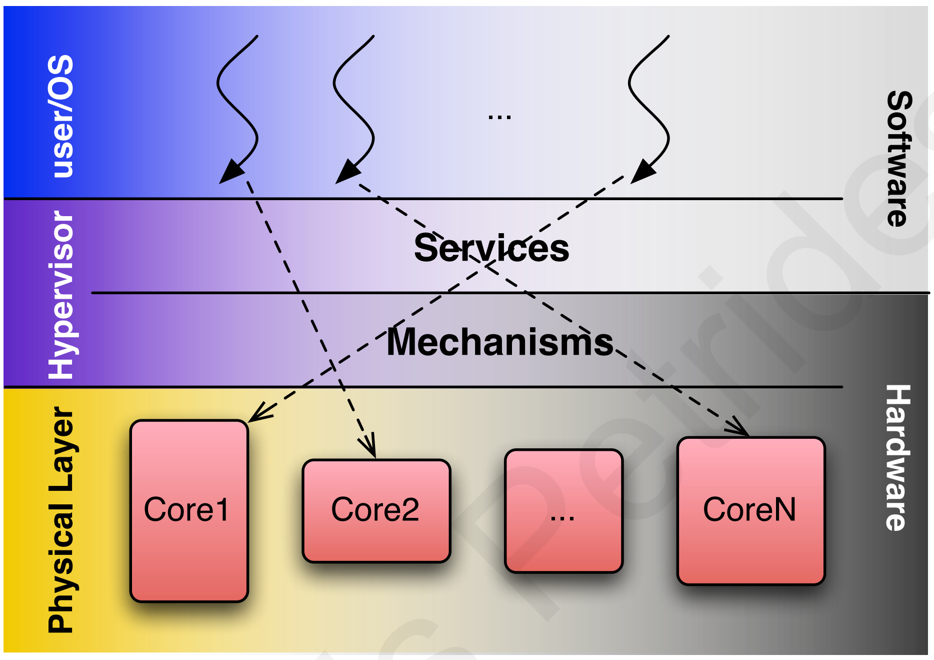


Figure 5.1: System layers.

5.2 Multi-core Architectures

5.2.1 Static Multi-core Configurations

As explained previously, the evolution of multi-core processors in the last few years allowed manufacturers to exploit new designs with different organizations. The organizations of modern multi-core architectures can be mainly classified as shown in Figure 5.2: symmetric homogeneous multi-cores, asymmetric homogeneous multi-cores and heterogeneous multi-cores.

In symmetric multi-core architectures, all the cores share the same characteristics (e.g. Instruction Set Architecture (ISA), hardware design). This architecture type is used in most modern architectures, such as the Intel and AMD CPUs, and latest NVIDIA GPUs, mostly because of how easy it is to design and use them. The increasing power consumption, the limited off-chip bandwidth [88], and in-

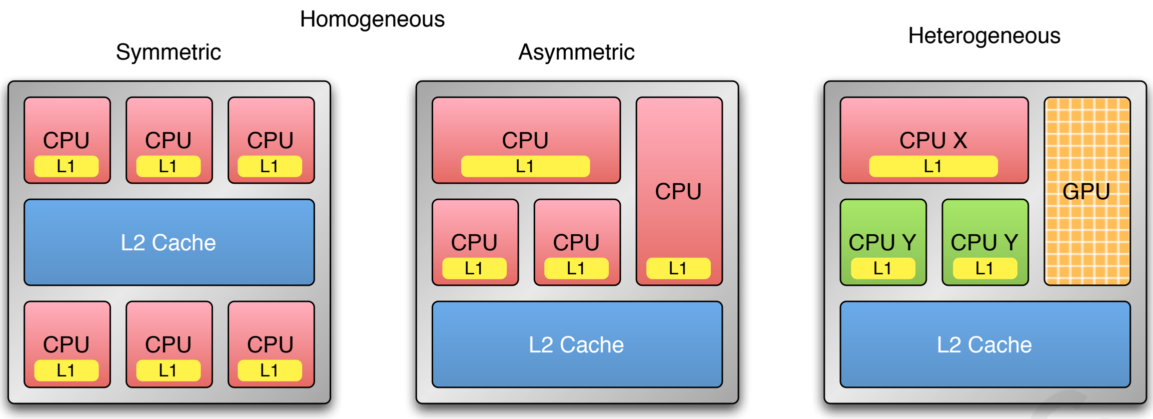


Figure 5.2: Classification of Static Architectures

trinsic technological issues (e.g., increase in permanent faults due to the technology down-scaling [89]), lead to different architectural configurations, such as the Nehalem [90] or the new Intel SCC architecture [4] which have groups of processing cores served by different memory controllers as an effort to increase the memory bandwidth per core and overcome the Bandwidth Wall limitation [91]. Other non-symmetric configurations combine cores with distinct capabilities to achieve a better application-to-hardware match, and consequently improving the performance-to-power ratio.

Asymmetric multi-cores are more attractive than heterogeneous because they do not require multiple binaries or dynamic binary translation, and can still provide enough flexibility to improve the application-to-hardware match. An example of an asymmetric system is the Intel SCC where the number of cores associated to each memory controller can vary, thus having groups of cores with different memory bandwidth capabilities. Examples of heterogeneous architectures include the IBM Cell/BE [16], the AMD fusion [92], and the GPUs when considered along with the host processor.

Without compromising the hypervisors' generality, this work is focused on asymmetric architectures and their main characteristics to design a Morphable multi-core that provide the best support to the upper layers, *i.e.*, applications, OS and hypervisor. Extending the proposed approach to support heterogeneous architectures with different ISAs should be achievable. Instead of demanding for N different binary codes, the hypervisor could be extended with an hardware translation mechanism such as the one used in Transmeta Crusoe [93].

Details of the proposed architecture

As previously mentioned, the proposed architecture falls into the category of asymmetric homogeneous multi-core systems. Thus, all the cores implemented share the same baseline ISA, which can be any canonical ISA such as x86. In addition, some cores may also include ISA extensions, e.g., for multimedia (such as MMX/SSE [94] or AltiVec [95]) and cryptography.

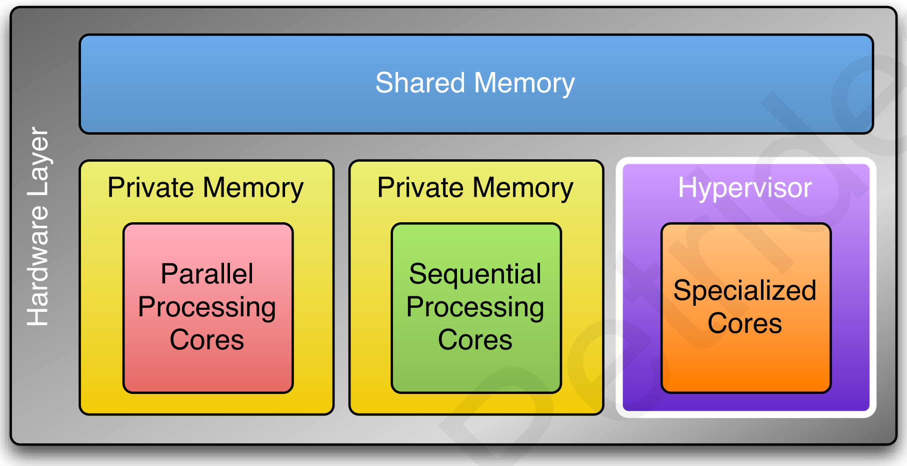


Figure 5.3: Static Architecture

The architecture consists of several cores with different functionalities as depicted in Figure 5.3, which combined in are able to improve the overall system performance. Since the hypervisor is general enough to manage architectures with different characteristics, there is not a strict definition of how the processor cores should be configured. However, considering in a broad sense the requirements of actual applications, the cores are divided into three large groups according to the roles they assume. The first two groups include general purpose cores for (a) parallel processing, and (b) sequential processing. The third group (c) specialized hardware, is used to support the Mechanisms provided by the hypervisor [96] (see Section 5.3).

The amount of parallelism contained in the different applications is the main factor that lead to distinguish between parallel and sequential processing cores. Sequential cores are mainly design to improve sequential execution, implementing techniques for example for aggressive ILP, while parallel processing cores are optimized for parallel execution. The latter cores are simpler and therefore it is possible to have more of them in the same circuit area, increasing the degree of parallelism. In addition, it is possible to also implement vector structures and other techniques to increase throughput. Both types of cores have access to two different types of memories: private memory and shared memory. This feature is built into the architecture because it reduces the overall memory latency, and makes the use of memory bandwidth more efficient. Also another strategy to increase the memory bandwidth supported per core is to have groups of cores served by different memory controllers.

The main purpose of the specialized hardware is to implement unique features to support the hypervisor Mechanisms.

The complexity of this hardware depends on the mechanisms supported, and moreover it requires an ISA extension in order to allow the hypervisor to directly interact with the specialized hardware and vice-versa. Also, because the ISA extension refers only for the specialized hardware, it does not interfere with the actual logical execution of an application, *i.e.* any program using the architecture does not require any changes in the binary code. Regarding the hypervisor Services, as explained in Section 5.3, the hypervisor should allocate general cores to execute these specific functions.

5.2.2 Dynamic Multi-core Configurations

The fact that applications behavior is unpredictable at hardware design time represents a limitation as it is not straightforward to define a priori the attributes of the several cores to be implemented. Reconfiguration is a way to surpass this limitation, and it appears naturally as a solution to dynamically reconstruct the architecture according to application requirements. Consequently, it is possible to improve the overall performance and at the same time have a more efficient architecture. However, reconfiguration introduces a new level of complexity into the system. Once more, the hypervisor can be used to encapsulate this additional complexity in a transparent way.

Reconfiguration can be performed at different granularities. Very fine-grain reconfiguration is not the best option in this case due the high reconfiguration time overheads.

Although a fully reconfigurable design allows to implement more efficient modules, these are also slower because the hardware requires complex routing mechanisms, and controllers due to the wide range of possible configurations.

Due to these overheads, this work is focused in a coarse-grain reconfigurable approach, which allows to configure only the most relevant parts of the architecture. In other words, the architecture may include a discrete set of configurations in a limited reconfiguration space, thus reducing the design complexity and overheads. Nevertheless, it is still flexible enough to adapt the architecture to the application requirements at run-time. The denomination ”morphable” designates those architectures which are not fully reconfigurable but are still able to adapt at run-time. Examples of other polymorphic architectures are the TRIPS [97], and the Core Fusion [98].

Finally, with the introduction of the dynamic configuration capabilities, the static architecture design provided in the previous section (Figure 5.3) is extended into the one shown in Figure 5.4. In addition to the specialized hardware mentioned before, the new architecture also requires a dedicated reconfiguration controller, which is used to support reconfiguration. Moreover, the configuration capabilities can be described by considering separately the computational elements, the on-chip memory hierarchy, and the memory controllers.

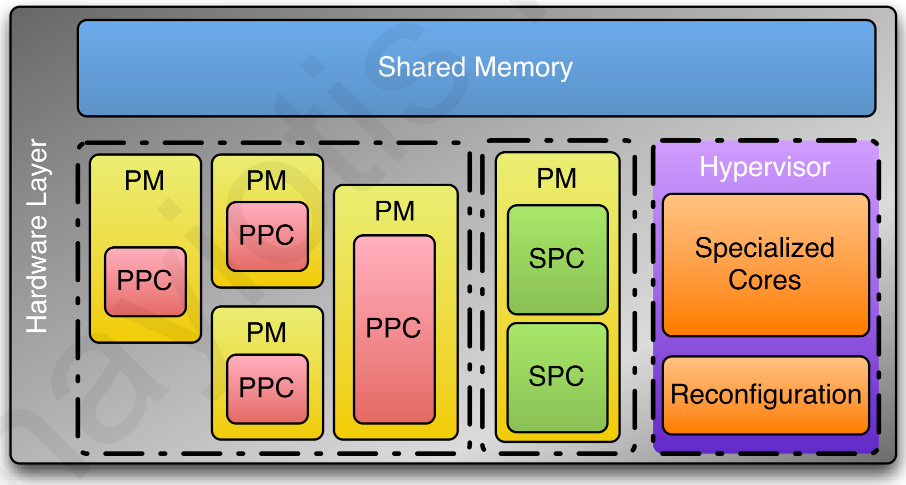


Figure 5.4: Morphable Architecture and its Components: Private Memory (PM), Sequential Processing Core (SPC), Parallel Processing Core (PPC)

Computational Element Reconfiguration

Reconfiguration or ”morphing” at the computational elements level is performed by selecting from of a fixed number of possible hardware configurations. The different configurations are obtained by performing reconfiguration at different levels: at the lower-level with the use hardware components that are reconfigurable, such as computational elements that may be merged [98], or memory elements that may change size, associativity or block size [99]; and at a higher-level where the voltage-frequency of the cores may be changed, or where some cores may be switched off [100].

Low-level reconfiguration is achieved by directly implementing configurable multi-core architectures, such as the Core Fusion [98]. This means that the proposed architecture can combine different embedded architectures that are already reconfigurable on their own. This only requires the adaptation of the reconfiguration specialized hardware to the embedded architectures. For example, if a massively parallel application is detected by the system, the hardware is configured to offer as much parallelism as possible by having simpler processing cores, and if there is the need for accelerating a specific computationally intensive thread, the cores can be "merged" to combine their processing power. On the other hand, configurations at a higher-level would concern particular architectural characteristics such as frequency, and number of active cores, computational units and memory.

Finally, adapting the frequency and the number of active cores at run-time according to the processing workload allows the processor to balance power consumption, performance, and thermal efficiency. For example if some cores are idle, their frequency can be reduced or they can be turned off to save power, or if for a certain period there is the need to accelerate a particular section of the execution, the frequency can be temporarily increased to improve performance. This type of reconfigurability works in a similar way to the power management system implemented on the IBM POWER6 architecture [100]

On-chip Memory Hierarchy and Reconfiguration

The design of the on-chip memory hierarchy is also of major importance to improve the architecture efficiency. In order to support a single address image to all cores in a multi-core processor, the on-chip memory can have at least one level of the hierarchy which is shared by all cores. This memory level may consists of a single memory module or by several separate modules (one per core in the extreme case), together with a coherence mechanism that ensures the correct update and access to the data shared in those modules.

When scaling the chip for a large number of cores, several factors affect the performance of the memory models as described above. First, designing a single large memory module is a difficult task but the most limiting factor will be the contention in the access by all cores to that single module. Second, even though partitioning that memory module into smaller pieces to be distributed among the different cores would solve the contention, the coherence mechanism results in a considerable overhead as the number of cores increases [101]. In addition, the behavior of the applications is different for different data structures. For example, in a parallel application certain data structures are shared among all threads while others are private to each thread. Mapping private data to the shared memory space may result in displacing useful data of another thread, and wasting resources for coherence where it is not required.

Considering all of the above it can be predicted that an efficient on-chip memory hierarchy for future large-scale processors will consist of both private and shared memory modules. The ratio between the sizes of these two spaces will depend from application to application and even from thread to thread. As such, it is proposed that caches should be able to be reconfigured at run-time, namely with respect to its line width, associativity and size. Thus, applications that require complex data management can benefit from the support provided by the shared memory, while applications with larger data parallelism that require higher memory bandwidth can benefit from the faster private memory. Also, when executing different independent applications on the same multi-core processor, there is no need to support a shared memory module between the different memory spaces.

Memory Controller Reconfiguration

As stated before, the increase of the number of cores on a chip leads to memory bandwidth limitations. Hardware solutions have been proposed to reduce this effect by servicing different processing cores by different memory controllers. The overall idea is to increase the available memory bandwidth per core. An example of such an architecture is the Intel SCC which contains 4 memory controllers, each one servicing a group of 12 cores.

This work proposes to go a step further and reconfigure, at run-time, the assignment of memory controllers to the different cores according to the application's demands. Thus, for a pool of available controllers these can be distributed to serve the processing cores according to the applications demands. For example if an application running on one core requires high memory bandwidth a controller can be assigned to serve only this core, while the other cores are served by the remaining controllers.

5.3 Virtualization of Morphables Multi-Cores

This work is focused on Bare-metal Virtualization [31, 102] and how its mechanisms and characteristics can be used in order to provide portability among different hardware configurations, and at the same time with low overhead for applications' execution. More specifically, the target of this work is to have a hypervisor, which will abstract the underlying hardware of the system from the OS and application layer.

The proposed platform targets systems composed by resources of different characteristics, and also systems whose characteristics can vary in time according to the applications. As stated in the previous section, the hypervisor will be designed in a modular way, with fundamental characteristics common to all systems. The only characteristic that will differ among different systems is the plug-ins and services supported for specific architectures. For example, for a reconfigurable system, the platform will provide the specific services in order to manage the reconfiguration of the devices (without disruption in the execution).

Finally, the functionality offered by the hypervisor should work transparently to the user/OS. The hypervisor has built-in mechanisms to monitor the behavior of the applications and thus take decisions accordingly. While these decisions may not be optimal, the fact that in this case the application is unaware of the architecture is a major benefit. Nevertheless, the hypervisor also provides a higher-level interface that allows the user/OS to control the usage of hypervisor's mechanisms. For example the user can fine-tune applications performance in order to exploit system's hardware characteristics at a more fine-grain level by triggering reconfiguration or assigning the executed threads to specific resources. Also the OS can use the offered features and extend them in order to improve scheduling decisions.

5.3.1 Hypervisor Mechanisms and Services

The functionalities of the hypervisor are implemented as a set of Mechanisms and Services. Figure 5.5 presents the Mechanisms and Services supported by the hypervisor and their detail description follows in the next paragraphs. By Mechanisms is defined the basic functionalities provided by the hypervisor in order to abstract the architecture to the upper layers in a transparent way.

Resource Detection Mechanism (RDM): This mechanism identifies the underlying hardware in terms of computational and memory features. Through this mechanism, the hypervisor will be aware of the current configuration of the underlying hardware and their performance capabilities. Typical information collected by this Mechanism includes the frequency and characteristics of each core, the memory cache configurations for each core and its peak performance. This information can be gathered by reading the performance counters [103] of the system. Another possible way to collect such information is by executing benchmarks composed by small kernels. This information is collected at the boot time and is stored into the Resource DataBase (RDB). The ISA extension allows upper layers to access the information collected by this mechanism as well as to trigger the update of the stored information when reconfiguration occurs. For example, in the case when cores dynamically change their frequency this mechanism must periodically update its information.

Threads Managing Mechanism (TMM): This mechanism is responsible for monitoring and migrating threads between cores. Threads' execution is monitored to gather statistics in terms of computation and memory demands, such as bandwidth utilization. The statistics data gathered are recorded in a Thread Database (TDB). Online monitoring techniques [63, 104] focus on monitoring the execution of an application through the hardware counters statistics in order to determine memory and computational demands of an application. For each thread the hypervisor organizes information for every interval of a given number of instructions, also named a phase, and stores the main statistics obtained (e.g. CPI, cache misses, and bandwidth) for the *N* last execution phases. The monitoring of the threads will provide the information needed in order to identify the characteristics of the thread. This information can then be used for improving the performance at run-time, and to make better scheduling decisions. For example, a bandwidth-aware scheduler would use this information to perform its decisions. This information can be gathered using the hardware performance counters [103, 105].

In order to accomplish that, a hardware performance counter per thread will be needed. Notice that in addition to the local cache information, the TMM needs also to monitor the cache coherence traffic and memory access patterns to identify possible access sharing and conflict patterns. This can also be achieved using the hardware performance counters as presented in [106]. Finally TMM also implements efficient schemes for migrating threads between cores.Thread migration from one core to another should occur with the minimal performance overhead (which maybe occur due to data transfer from the already executing core to the new assigned core) and be transparent among other threads, in order not to influence their execution.

Core Monitoring Mechanism (CMM): This mechanism monitors cores' resource utilization, temperature, idle time of the core, working frequency, power consumption through the hardware performance counters.

Thread Recognition Mechanism (TRM): The data stored in the TDB is used by the TRM to categorize the different Threads and identify them using a Thread Template (TT), which represents a description of the Thread's behavior according to its memory, computational and bandwidth demands.

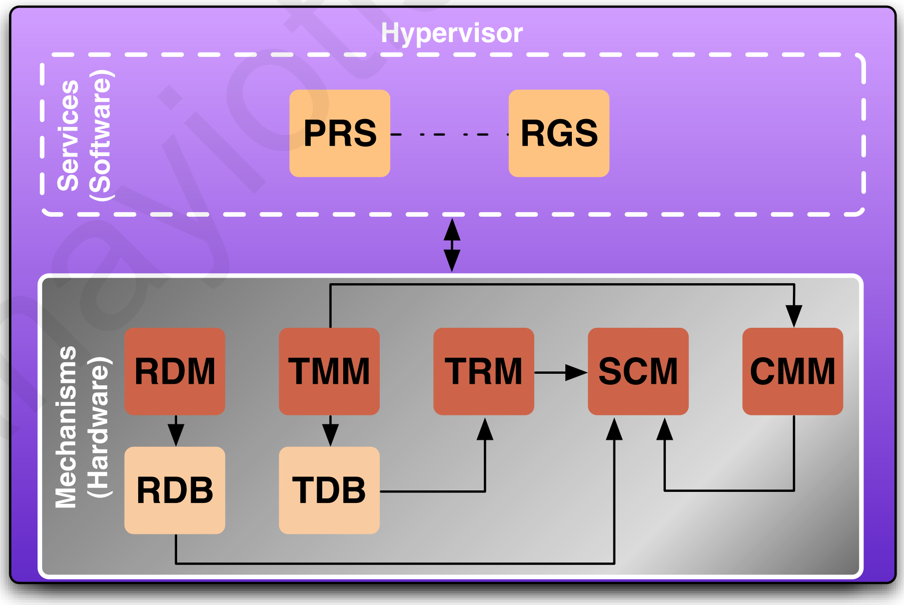


Figure 5.5: Hypervisor Architecture.

The Services implement functionalities to facilitate the hypervisor usage, or to improve the performance of the applications. The Services include support for general operations such as prefetching and scheduling schemes, and functionalities that are

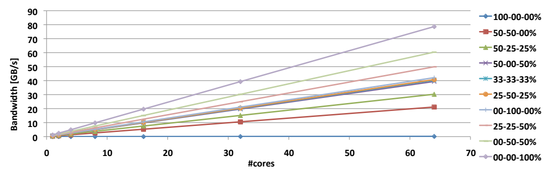


Figure 5.6: Bandwidth demands for short executing applications.

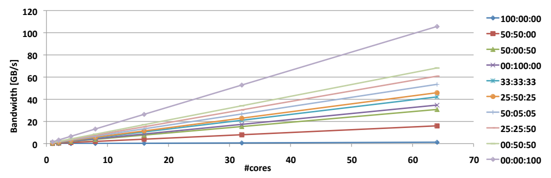


Figure 5.7: Bandwidth demands for long executing applications.

dedicated to the specific underlying hardware, such as reconfiguration. The user/OS has the option of using the provided Services or implementing their own extensions. Finally the Services are supported by the previous described Mechanisms.

Low-level Scheduling Service (SCS): The Low-level Scheduling Service, is a baseline scheduling engine that can map the application threads to the architecture processing cores according to their demands. It uses online monitoring information, namely information collected by the several mechanisms offered in the system, in order to decide which core fits best for the execution of threads. Many studies were focused on how to schedule applications' threads in order to achieve high levels of performance using information obtained by offline and online monitoring techniques [63, 65, 104].

In this work the hypervisor uses online information provided by the RDM, TMM, CMM and TRM mechanisms, in order to determine the most suitable run-time scheduling assignment of the threads for each particular application [107, 108]. As Bower [109] has shown, the scheduling of an applications' threads must take into consideration the current state of each core in order to achieve the desired performance levels.

The data sharing information of the executing threads, provided by the TRM, can be used to take better decisions such as guiding the scheduling of threads done by the OS to more appropriate resources.

For example, in a multi-core where the L2 is shared pair-wise such as the Intel Quad-core Xeon processor, two sharing threads should be scheduled to the cores sharing the L2 (which do not suffer from conflicting misses) while conflicting threads should be scheduled to cores having different L2 caches. The scheduler should also be able to identify certain types of applications which can be directly mapped to specific types of cores. For example, parallel threads can be directly pinned to parallel processing cores.

In more details, regarding the bandwidth-aware scheduler, different applications according both to their execution time and their bandwidth demands are profiled. From the preliminary results applications are able to be categorized according to their execution time to: short, medium and long, and according to their bandwidth demands to: low, medium and high bandwidth demanding applications. Figures 5.6 and 5.7 present for short and long executed applications the bandwidth demands of different applications' combinations of different bandwidth demands. More specifically, the X%- Y% - Z% notation resembles low, medium and high bandwidth demanding applications ratio executing simultaneously on the system. It can be observed from the results depicted that the bandwidth demand increases as the number of cores increases and that the different combinations of applications also suppress the off-chip bandwidth. From the analysis of these results it can be shown that if the bandwidth requirements of the applications are not satisfied there is a high impact to the applications performance. More specifically, if there is no bandwidth utilization policy among the available memory controllers assigned to the cores of the system, impact to applications performance can be as high as slowing down more than X times. If a bandwidth-aware scheduling policy is applied, where the bandwidth of the memory controllers is utilized and no under-utilization or over-utilization is observed among controllers, preliminary results has shown that for both short and long executing applications a performance improvement of almost 2X for most of the applications combinations can be achieved. It is important to mention also that the bandwidth utilization of the controllers assigned to the different cores can be achieved by balancing the applications, or the threads, that are executed to the cores of each assigned controller. Migrating the applications between cores served from different controllers can be achieved by monitoring their execution according to their bandwidth demands.

A practical example would be the case where a bandwidth-aware scheduler is implemented on the Intel SCC architecture [4] to distribute the application threads by processing cores served by different memory controllers according to their demands. As stated before, memory bandwidth has become a limitation with the increasing number of cores in recent architectures and it will be important for future architecture to have efficient scheduling mechanism that take advantage of the applications' different memory bandwidth requirements to improve the overall system performance. Namely, the scheduling mechanism takes into consideration the fact that different processing cores are served by different memory controllers and thus assign applications to different cores trying to balance the applications requirements and the overall memory bandwidth capabilities. In this case the hypervisor would use the TMM to obtain the bandwidth requirements of each thread for every phase, and according to that information and the placement of the threads at that given instant, if a bandwidth violation is detected, *i.e.*, if the maximum bandwidth supported by the controlled responsible for a given group of threads is violated, the scheduler tries to redirect some of the threads to a different controller with smaller bandwidth utilization. This redirection is performed using TMM, to migrate the threads to a different core served by another controller, or if in a system with reconfigurability capabilities, the core may just be reconnected to a different memory controller using the Reconfiguration Service.

Prefetching Services (PRS): Many applications have irregular memory access patterns that can not be captured neither by the compiler as they may depend on dynamically determined values, nor by prefetching engines as they are irregular. Thus, a careful monitoring of the data accesses may result in identifying non trivial memory access patterns. The streams of data are stored to save the effort of identifying these data streams. Based on these streams, an intelligent prefetching identifies irregular memory access patterns and triggers the prefetching service in order to increase applications performance.

Among others, Papadopoulos *et al.* [110] proposed a similar system. This prefetching should be done transparently to the OS and application. Information provided by the thread template recognition mechanism to the platform, *i.e.* memory access patterns, will trigger the prefetching service.

Reconfiguration Services (RES): This service is relevant when the underlying hardware has the ability to change its configuration dynamically at run-time, namely to control the three types of reconfigurable capabilities described in Section 5.2.

The main difference between this Service and the one presented before is that in order to perform the hardware reconfigurations, there is a need to have a reconfiguration controller which must be implemented in hardware [111]. The RES will work very closely with the SCS: before a thread is scheduled, ideally the system would be reconfigured for the best match between thread demands and available hardware. The reconfiguration of the cores may be performed in two different ways: (i) user/OS reconfiguration requests and (ii) hypervisor reconfiguration. In

(i) the user/OS has the privilege to determine the configuration of the system for the application as to exploit its characteristics and the system's resources, while in (ii) the hypervisor automatically determines the configuration that achieves the best performance for the running applications by using the statistics inferred from the monitoring mechanisms. Overall, the reconfiguration of the systems' components is achieved by modifying registers or memory positions reserved for that purpose. These registers can be accessed by the user/OS through specific ISA extensions. As a practical example lets consider again the bandwidth-aware scheduler, and that a certain moment it detects a memory bandwidth violation (see SCS description). The SCS may take two possible actions to compensate for the bandwidth violation, one is to migrate the thread responsible for the violation to another core, which is served by a different controller. However, a second strategy is to use the RES to reconfigure the controller connections in order to redirect the traffic to another controller without actually migrating the threads. The system may decide which solution is the best for every given case depending on the overheads associated, *i.e.*, in the first case the latency due to re-cacheing and other effects of migrating the threads to a different core, and in the second case the latency of performing the hardware reconfiguration.

5.4 Integration of the Proposed Techniques

In the previous chapters different techniques have been presented which fit the vision of the modular virtualization layer for supporting such architectures. More specifically, the Heterogeneous NUMA-aware scheduling policy presented in Chapter 4, includes a number of mechanisms as described in this section. These mechanisms are: Resource Detection, Thread Monitoring and Thread Migration. Therefore, they can be integrated to the hypervisor and provide the desired functionalities since they are portable as presented in Chapter 4. Moreover, the scheduling policy itself can be seen as a module of the Threads Migration Mechanism, which will consider applications demands in terms of resources and assign a best matching resource for their execution. Additionally, the data prefetching technique presented in Chapter 2 can be integrated as the extended hypervisor mechanism for prefetching. The presented techniques and their integration to the proposed modular virtualization layer will result on a whole system which will manage both the underlying hardware and the executed applications. Its target is both single and multi-application performance improvement. As presented in Chapter 3, the proposed virtualization layer adds very limited overhead to the performance of the executed applications. Moreover, exploiting the isolation properties provided by the virtualization techniques, resources can be logically divided into different Domains by a Domain Service to provide performance predictability of co-executed applications.

5.5 Summary

Future multi-core processors will be composed of cores with different computational and memory capabilities, which are also able to change their configuration at run-time. In this chapter tries to address the issue of managing and exploiting such future large-scale systems not only in terms of software, by means of a virtualization approach proposing for that an hypervisor, but also the hardware architecture and design. A Virtualization Platform is proposed, *i.e.* a complete system able to wrap the complexity of the underlying hardware, through a hypervisor module. In addition, the design a Morphable multi-core is proposed, where its resources are managed by the hypervisor to transparently tune and achieve an improvement of the overall system efficiency.

The hypervisor proposed in this chapter manages applications' threads transparently to the OS and the applications. It supports a set of Mechanisms and Services, which are supported by specialized hardware. Moreover, the fact that the system is able to adapt at run-time to the applications' demands releases the programmer from knowing the details of the architecture. However, the user and/or OS has the possibility to implement hardware-aware applications using ISA extensions that give access to the implemented Mechanisms in order to fine-tune and improve applications' performance levels. Moreover, the proposed Morphable multi-core is composed of several asymmetric cores which have the capability of changing their configuration at different levels for both their logic and memory elements. This results in a more efficient hardware platform that besides being able to adapt the application execution to the hardware underneath, is also able to adapt the hardware to the demands of the different applications and/or phases. Although this two-way adaptation increases the adaptability of the overall system at the cost of some additional complexity, it is still handled in a transparent way by the proposed hypervisor.

Chapter 6

Conclusions and Future Work

Achieving high levels of application performance on a many-core architecture environment considering the characteristics of both available resources and applications demands is not a trivial task. Different approaches exist in order to exploit the increasing number of resources and at the same time target on high performance of applications. In particular, one approach is to target single application performance by exploiting its own parallelism while another approach is to target multiple application performance by exploiting throughput parallelism. Both approaches result in different challenges. This thesis is focused on these challenges which are identified as: *(i)* tuning single application performance by considering both many-core underlying resources and application characteristics, *(ii)* minimizing interference between co-executing applications and *(iii)* satisfying the dynamic demands of applications when executing on a clustered heterogeneous many-core environment.

6.1 Achieved Objectives and Contributions

To achieve high levels of scalability and efficiency of memory demanding applications exploiting many-core architecture characteristics (Objective 1 - Parallelism on a Clustered Many-Core Architecture), three different queries are ported from the TPC-H benchmark suite on the Intel SCC experimental processor. Their performance behaviour is studied when data prefetching is applied using the on-chip shared memory of the system. Experiments depict that when there is no data reusage on the query algorithms (Q6) data prefetching shows no significant improvement. For medium complexity query algorithm with high input data size (Q12) nested-loop join algorithm using data prefetching can achieve up to 5x speedup. Although in this case hash join implementation is more efficient due to the simplicity of its algorithm. Finally for high complex queries in terms of the operations performed and high input data size (Q3) using a hybrid implementation of hash join and nested-loop join with data prefetching it is posssible to improve performance by a factor *10*. Additionally, the power-performance efficiency of the different queries implementations is investigated for the most efficient implementation. Results show that in the case of simple query algorithms, like Q6 and Q12 hash join implementation, scaling down the systems' cores frequency and reducing the number of cores (executing the respective implementation) can achieve both high power-performance efficiency and throughput when executing the query in multiple instances on the system.

In order to offer performance guarantees for the co-execution of multiple high-performance computing applications on many-core systems without adding significant overheads (Objective 2 - Guarantee Performance), the performance overhead and performance isolation is analyzed of parallel applications while executing on top of different virtualization environments. Results show that for most applications the overhead is relatively small. Virtualization shows to be an important tool as to create performance domains where the performance of HPC applications can be safeguard, independent of the applications executing on the rest of the multi-core processor. Also, it is important to mention that by using vitualization not only predictable performance is achieved but also the interference of applications co-executing is limited on the virtualized system compared to their co-execution on the native system. It can be also observed that for more than half of the presented experiments the execution on the bare-metal virtualized system achieved better performance than the native execution. Moreover, the overhead observed for the execution of the applications was only up to 3%. Performance isolation provided by virtualization, allows applications to achieve their predicted speedup without suffering any interference from co-execution of other applications on the same processor at the same time. This is becoming more relevant for large-scale multi-core processors on which full utilization is desired by running simultaneously different applications. Overall the results were very encouraging for the use of virtualization for future large-scale multi-core processors even for demanding HPC applications.

Given the application, determining appropriate resources for power-performance efficiency is not a trivial task. In addition, given the architecture characteristics of a clustered heterogeneous many-core architecture it is very difficult in practice to coordinate the scheduling operations during runtime. The key for determining the best matching core for a certain application is to find out the application's memory and computational requirements. These findings had led to the proposed scheduling policy which dynamically find the best matching resources for multiple high-performance applications on a heterogeneous clustered many-core system (Objective 3 - Heterogeneous and NUMA-aware Scheduling) considering the non-uniform memory latency, the heterogeneity of the cores, and the contention to the memory controller to find the best matching core for the application's memory and compute requirements. Results of the proposed scheduler show that satisfying applications requirements during their execution can improve significantly their performance. In particular, compute-bound applications can improve their execution time up to 36% and memory-bound applications up to 15%. Moreover, the experimental results show that the energy efficiency of the system is improved, of about 40%, and the same time achieve a very good performance, of about 18%, for compute-bound applications. Given the priority of the proposed policy, memory-bound applications show smaller improvements in both performance and power consumption. The results are very encouraging for the use of such feature-aware assignment policies for future many-core processors.

6.2 Open Research Questions

The results of this thesis show that in order for applications to exploit the performance benefits of multiple heterogeneous cores in a system there is a need of a runtime environment that can help with different tasks such as data prefetching, performance isolation between co-execution, and best matching of resources determined dynamically. At the same time, new research directions arise from the outcome of this work.

6.2.1 Direction 1: Machine Learning Algorithms

This thesis proposes a runtime system which considers the non-uniform memory latency, the heterogeneity of the cores, and the contention to the memory controller to find the best matching core for the application's memory and compute requirements on a clustered many-core system. This proposed runtime could be further enhanced with machine learning algorithms features which could identify similar behaviours of applications execution and find a best matching resource more effectively. The target of such enhancement will be to recognize applications behaviour during execution and satisfy their demands throughout execution. Additionally, recognizing execution patterns will allow the system to recognize future needs of applications and apply near optimal placement through application placement planning. The runtime though needs to be adapted in order to recognize patterns of applications execution and at the same time not to be intrusive to applications.

6.2.2 Direction 2: Dynamic Heterogeneous Many-Cores

This thesis studied static clustered many-core heterogeneous architectures, having their configuration predefined and static through the whole execution of applications. In the future it can be considered that resources within a many-core architecture can change their configuration dynamically, *i.e.* core frequency. Initiating the change of resource configuration dynamically through the runtime can further improve its effectiveness. Changing resources configurations dynamically can result in forming performance domains suitable for applications execution and at the same time achieve higher power-performance utilization of the system. Additionally, different targets may be set for the runtime such as high performance levels or efficient power-performance efficiency.

6.2.3 Direction 3: Fault-tolerance

Another direction of this work could be extending the functionality of the proposed runtime system for clustered many-core architectures to incorporate fault-tolerance and how it should react in case of failures of the hardware. Failures on large-scale many-core systems it is possible to occur either caused by hardware or by disabling cores. In such a case, the runtime should be able to react and mitigate its consequences by rescheduling applications to other available resources. This enhancement will increase the reliability of a system, ensuring that applications execution will not failed due to hardware faults.

6.2.4 Direction 4: Morphable Many-Cores Runtime System

Incorporating all proposed mechanisms presented in this thesis under a runtime system for morphable many-core architectures is another direction of this work. More specifically, each mechanism presented can be seen as a service of a runtime system targeting morphable many-core architectures. Additionally, the level of reconfiguration of the underlying hardware can be investigated enriching the runtime effectiveness and mitigating the challenges raised from the static many-core architectures.

Bibliography

[1] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson, "A 48-core ia-32 message-passing processor with dvfs in 45nm cmos," in *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*, Feb 2010, pp. 108–109.

[2] I. A. C. Ureña, M. Riepen, and M. Konow, "Rckmpi - lightweight mpi implementation for intel's single-chip cloud computer (scc)," in *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 208–217. [Online]. Available: http://dl.acm.org/citation.cfm?id=2042476.2042500

[3] Transaction Processing Council, "TPC Benchmark H (Decision Support) Standard Specification, Revision 2.6.1," June 2006.

[4] Intel, "Single-chip Cloud Computer," http://techresearch.intel.com/UserFiles/en-us/File/terascale/SCC-Overview.pdf, 2009.

[5] P. Petrides, A. Diavastos, and P. Trancoso, "Exploring decision support queries on futured many-core architectures," in *3rd Many-core Applications Research Community (MARC) Symposium. Proceedings of the 3rd MARC Symposium, Ettlingen, Germany, July 5-6, 2011*. KIT Scientific Publishing, Karlsruhe, 2011, pp. 81–84. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023937

[6] P. Petrides, A. Diavastos, C. Christofi, and P. Trancoso, "Scalability and efficiency of database queries on future many-core systems," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb 2013, pp. 24–28.

[7] P. Petrides, G. Nicolaides, and P. Trancoso, "Hpc performance domains on multi-core processors with virtualization," in *Architecture of Computing Systems – ARCS 2012*, A. Herkersdorf, K. Römer, and U. Brinkschulte, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 123–134.

[8] P. Petrides and P. Trancoso, "Addressing the challenges of future large-scale many-core architectures," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '13. New York, NY, USA: ACM, 2013, pp. 6:1–6:4. [Online]. Available: http://doi.acm.org/10.1145/2482767.2482776

[9] P. Petrides and P.Trancoso, "Heterogeneous- and numa-aware scheduling for many-core architectures," in *Proceedings of the 10th ACM International Systems and Storage Conference*, ser. SYSTOR '17. New York, NY, USA: ACM, 2017, pp. 2:1–2:12. [Online]. Available: http://doi.acm.org/10.1145/3078468.3078482

[10] P. Petrides, F. Pratas, L. Sousa, and P. Trancoso, "Virtualization for morphable multi-cores," in *Proceedings of the 2nd Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures (PARMA) (co-located with ARCS 2011)*, February 2011, pp. 137–143.

[11] P. Trancoso, D. Othonos, and A. Artemiou, "Data parallel acceleration of decision support queries using Cell/BE and GPUs," in *Proceedings of the 6th ACM conference on Computing frontiers*. ACM New York, NY, USA, 2009, pp. 117–126.

[12] T. Sherwood, S. Sair, and B. Calder, "Predictor-directed stream buffers," in *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, 2000, pp. 42–53.

[13] D. Koufaty and J. Torrellas, "Compiler support for data forwarding in scalable shared-memory multiprocessors," in *Proceedings of the 1999 International Conference on Parallel Processing*, 1999, pp. 181–190.

[14] K. Papadopoulos, K. Stavrou, and P. Trancoso, "Helpercoredb: Exploiting multicore technology to improve database performance," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–11.

[15] C.-K. Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *Proceedings 28th Annual International Symposium on Computer Architecture*, 2001, pp. 40–51.

[16] J. A. Kahle *et al.*, "Introduction to the Cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4/5, p. 589, 2005.

[17] M. Monteyne and R. Inc, "RapidMind Multi-Core Development Platform," *Rapid-Mind, Tech. Rep*, 2007.

[18] M. W. van Tol, R. Bakker, M. Verstraaten, C. Grelck, and C. R. Jesshope, "Efficient memory copy operations on the 48-core intel scc processor," in *Many-core Applications Research Community (MARC) Symposium*, 2011, pp. 13–18.

[19] K. Avdic, N. Melot, C. Kessler, and J. Keller, "Pipelined parallel sorting on the intel scc," in *Fourth Swedish Workshop on Multicore Computing (MCC)*, 2011, pp. 96–101.

[20] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal, "Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs," in *SC 2006 Conference, Proceedings of the ACM/IEEE*, Nov 2006, pp. 14–14.

[21] C. Isci, A. Buyuktosunoglu, C. y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Dec 2006, pp. 347–358.

[22] N. Ioannou, M. Kauschke, M. Gries, and M. Cintra, "Phase-based application-driven hierarchical power management on the single-chip cloud computer," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, Oct 2011, pp. 131–142.

[23] P. Thanarungroj and C. Liu, "Power and energy consumption analysis on intel scc many-core system," in *30th IEEE International Performance Computing and Communications Conference*, Nov 2011, pp. 1–2.

[24] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core scc processor: the programmer's view," in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2010, pp. 1–11.

[25] H. Vandierendonck and P. Trancoso, "Building and validating a reduced tpc-h benchmark," in *14th IEEE International Symposium on Modeling, Analysis, and Simulation*, Sept 2006, pp. 383–392.

[26] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens, "Quantifying the performance isolation properties of virtualization systems," in *Proceedings of the 2007 Workshop on Experimental Computer Science*, ser. ExpCS '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1281700.1281706

[27] F. Rodríguez, F. Freitag, and L. Navarro, "On the use of intelligent local resource management for improved virtualized resource provision: challenges, required features, and an approach," in *HPCVirt '08: Proceedings of the 2nd workshop on System-level virtualization for high performance computing*. New York, NY, USA: ACM, 2008, pp. 24–31.

[28] Amazon, "Amazon Elastic Compute Cloud: Getting Started Guide," http://docs.amazonwebservices.com/AWSEC2/latest/GettingStartedGuide/, 2009.

[29] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, October 2008, pp. 72–81.

[30] Sun, "Sun Microsystems VirtualBox," http://www.virtualbox.org/, 2010.

[31] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177. [Online]. Available: http://doi.acm.org/10.1145/945445.945462

[32] C. Macdonell and P. Lu, "Pragmatics of virtual machines for high-performance computing: A quantitative study of basic overheads," 2007.

[33] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the xen virtual machine environment," in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, ser. VEE '05. New York, NY, USA: ACM, 2005, pp. 13–23. [Online]. Available: http://doi.acm.org/10.1145/1064979.1064984

[34] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 275–287. [Online]. Available: http://doi.acm.org/10.1145/1272996.1273025

[35] L. Youseff, R. Wolski, B. Gorda, and C. Krintz, "Paravirtualization for hpc systems," in *Proceedings of the 2006 International Conference on Frontiers of High Performance Computing and Networking*, ser. ISPA'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 474–486. [Online]. Available: http://dx.doi.org/10.1007/11942634_49

[36] W. Huang, J. Liu, B. Abali, and D. K. Panda, "A case for high performance computing with virtual machines," in *ICS '06: Proceedings of the 20th Annual International Conference on Supercomputing*. New York, NY, USA: ACM, 2006, pp. 125–134.

[37] H. Payer, H. Röck, and C. M. Kirsch, "Get what you pay for: Providing performance isolation in virtualized execution environments," 09 2010.

[38] T. E. Carlson, W. Heirmant, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2011, pp. 1–12.

[39] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: Challenges in and avenues for cmp scaling," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 371–382. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555801

[40] N. Ioannou, M. Kauschke, M. Gries, and M. Cintra, "Phase-based application-driven hierarchical power management on the single-chip cloud computer," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 131–142. [Online]. Available: http://dx.doi.org/10.1109/PACT.2011.19

[41] A. Jaleel, "Memory Characterization of Workloads Using Instrumentation-Driven Simulation," Retrieved from http://www.glue.umd.edu/ ajaleel/workload/.

[42] G. Long, D. Fan, and J. Zhang, "Characterizing and understanding the bandwidth behavior of workloads on multi-core processors," in *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 110–121.

[43] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker, "Scalable thread scheduling and global power management for heterogeneous many-core architectures," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 29–40. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854283

[44] D. Feitelson, L. Rudolph, and U. Schwiegelshohn, "Parallel job scheduling - a status report," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Springer Berlin Heidelberg, 2005, vol. 3277, pp. 1–16. [Online]. Available: http://dx.doi.org/10.1007/11407522_1

[45] D. G. Feitelson, "Job Scheduling in Multiprogrammed Parallel Systems," IBM Research Report 19790, Tech. Rep., 1997.

[46] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 306–318, 1992.

[47] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488

[48] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633

[49] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A resource allocation model for qos management," in *Proceedings Real-Time Systems Symposium*, Dec 1997, pp. 298–307.

[50] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle, "Managing energy and server resources in hosting centers," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '01. New York, NY, USA: ACM, 2001, pp. 103–116. [Online]. Available: http://doi.acm.org/10.1145/502034.502045

[51] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson, "Automatic exploration of datacenter performance regimes," in *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, ser. ACDC '09. New York, NY, USA: ACM, 2009, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/1555271.1555273

[52] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed job latency in data parallel clusters," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012, pp. 99–112. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168847

[53] F. Sironi, D. Bartolini, S. Campanoni, F. Cancare, H. Hoffmann, D. Sciuto, and M. Santambrogio, "Metronome: Operating system level performance management via self-adaptive computing," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, June 2012, pp. 856–865.

[54] M. Otoom, A. Jaleel, and P. Trancoso, "Using personality metrics to improve cache interference management in multicore processors," in *Proceedings of the 14th ACM international Conference on Computing Frontiers*, ser. CF '17, 2017, pp. 1–4.

[55] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 248–259. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155650

[56] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, "An analysis of performance interference effects in virtual environments," in *2007 IEEE International Symposium on Performance Analysis of Systems Software*, April 2007, pp. 200–209.

[57] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 283–294. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000099

[58] A. Verma, P. Ahuja, and A. Neogi, "Power-aware dynamic placement of hpc applications," in *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ser. ICS '08. New York, NY, USA: ACM, 2008, pp. 175–184. [Online]. Available: http://doi.acm.org/10.1145/1375527.1375555

[59] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang, "Online cache modeling for commodity multicore processors," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 4, pp. 19–29, Dec. 2010. [Online]. Available: http://doi.acm.org/10.1145/1899928.1899931

[60] J. M. Calandrino, "On the design and implementation of a cache-aware soft real-time scheduler for multicore platforms," Ph.D. dissertation, Chapel Hill, NC, USA, 2009, aAI3366308.

[61] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 47–58. [Online]. Available: http://doi.acm.org/10.1145/1272996.1273004

[62] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen, "Processor hardware counter statistics as a first-class system resource," in *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, ser. HOTOS'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 14:1–14:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1361397.1361411

[63] A. Fedorova, D. Vengerov, and D. Doucette, "Operating system scheduling on heterogeneous core systems," in *Proceedings of 2007 Operating System Support for Heterogeneous Multicore Architectures*, 2007.

[64] A. Fedorova, S. Blagodurov, and S. Zhuravlev, "Managing contention for shared resources on multicore processors," *Commun. ACM*, vol. 53, no. 2, pp. 49–57, Feb. 2010. [Online]. Available: http://doi.acm.org/10.1145/1646353.1646371

[65] D. Shelepov and A. Fedorova, "Scheduling on heterogeneous multicore processors using architectural signatures," in *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, 2008, pp. 21–25.

[66] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "Hass: A scheduler for heterogeneous multicore systems," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 66–75, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1531793.1531804

[67] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/1362622.1362694

[68] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proceedings of the 3rd Conference on Computing Frontiers*, ser. CF '06. ACM, 2006, pp. 29–40. [Online]. Available: http://doi.acm.org/10.1145/1128022.1128029

[69] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-isa heterogeneous multi-core architectures for multithreaded workload performance," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 64–75. [Online]. Available: http://dl.acm.org/citation.cfm?id=998680.1006707

[70] A.-H. Haritatos, G. Goumas, N. Anastopoulos, K. Nikas, K. Kourtis, and N. Koziris, "Lca: A memory link and cache-aware co-scheduling approach for cmps," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 469–470. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628123

[71] W. Heirman, T. E. Carlson, K. V. Craeynest, I. Hur, A. Jaleel, and L. Eeckhout, "Undersubscribed threading on clustered cache architectures," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 678–689.

[72] M. Kaliorakis, M. Psarakis, N. Foutris, and D. Gizopoulos, "Accelerated online error detection in many-core microprocessor architectures," in *2014 IEEE 32nd VLSI Test Symposium (VTS)*, April 2014, pp. 1–6.

[73] Tilera, "Tile-MX Multicore Processor," http://www.tilera.com.

[74] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, September 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[75] P. Petrides, F. Pratas, L. Sousa, and P. Trancoso, "Exploiting location-aware task execution on future large-scale many-core architectures," Technical Report TR-12-4, University of Cyprus, Department of Computer Science, 2012.

[76] CryoPID, "A Process Freezer for Linux," http://cryopid.berlios.de/.

[77] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The nas parallel benchmarks," *Int. J. High Perform. Comput. Appl.*, vol. 5, no. 3, pp. 63–73, Sep. 1991. [Online]. Available: http://dx.doi.org/10.1177/109434209100500306

[78] Intel, "Intel Xeon Phi Coprocessor," https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner.

[79] I. Labs, "History of Many-Core Leading to Intel Xeon Phi," http://download.intel.com/newsroom/kits/xeon/phi/pdfs/Many-Core-History_Backrounder.pdf.

[80] ARM, "big.LITTLE Technology," https://www.arm.com/products/processors/biglittleproc

[81] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, July 2008.

[82] F. Pratas, P. Trancoso, A. Stamatakis, and L. Sousa, "Fine-grain parallelism using multi-core, cell/be, and gpu systems: Accelerating the phylogenetic likelihood function," in *2009 International Conference on Parallel Processing*, Sept 2009, pp. 9–17.

[83] L. Dagum and R. Menon, "Open MP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[84] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[85] Intel, "Parallel Studio," http://software.intel.com/en-us/intel-parallel-studio-home/.

[86] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou, "Ct: A flexible parallel programming model for tera-scale architectures," *Intel Technology Journal*, vol. 11, no. 4, October 2007.

[87] A. Munshi, "OpenCL: Parallel computing on the GPU and CPU," *SIGGRAPH, Tutorial*, 2008.

[88] B. Rogers, G. Bell, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: challenges in and avenues for CMP scaling," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ACM New York, NY, USA, 2009, pp. 371–382.

[89] H. Iwai, "Technology roadmap for 22nm and beyond," in *2009 2nd International Workshop on Electron Devices and Semiconductor Technology*, June 2009, pp. 1–4.

[90] Intel, "Intel 64 and IA-32 Architectures Optimization Reference Manual," http://developer.intel.com/products/processor/manuals/, 2008.

[91] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: challenges in and avenues for CMP scaling," in *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 2009, pp. 371–382.

[92] A. Devies, "The future is fusion: The Industry-Changing Impact of Accelerated Computing," *Advanced Micro Devices*, 2008.

[93] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing[a] Software: using speculation, recovery, and adaptive retranslation to address real-life challenges," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society Washington, DC, USA, 2003, pp. 15–24.

[94] A. Strey and M. Bange, "Performance Analysis of Intel's MMX and SSE: A Case Study," *Lecture Notes in Computer Science*, pp. 142–147, 2001.

[95] J. Tyler, J. Lent, A. Mather, and H. Nguyen, "AltiVec[a]: Bringing vector technology to the PowerPC[a] processor family," in *IEEE Int. Conf. Performance, Computing Communications*, 1999, pp. 437–444.

[96] D. Upton and K. Hazelwood, "Heterogeneous Chip Multiprocessor Design for Virtual Machines," in *2nd Workshop on Software Tools for Multicore Systems (STMCS)*, Mar. 2007.

[97] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. Keckler, and C. Moore, "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2, pp. 422–433, 2003.

[98] E. Ipek, M. Kirman, N. Kirman, and J. Martinez, "Core fusion: accommodating software diversity in chip multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ACM, 2007, p. 197.

[99] M. Qureshi, D. Thompson, and Y. Patt, "The V-way cache: Demand based associativity via global replacement," in *Annual International Symposium on Computer Architecture*, vol. 32. IEEE Computer Society 1999, 2005, p. 544.

[100] H.-Y. McCreary, M. A. Broyles, M. S. Floyd, A. J. Geissler, S. P. Hartman, F. L. Rawson, T. J. Rosedahl, J. C. Rubio, and M. S. Ware, "Energyscale for ibm power6 microprocessor-based systems," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 775–786, Nov. 2007.

[101] D. Lilja, "Cache coherence in large-scale shared-memory multiprocessors: Issues and comparisons," *ACM Computing Surveys (CSUR)*, vol. 25, no. 3, pp. 303–338, 1993.

[102] J. Smith and R. Nair, *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann Pub, 2005.

[103] F. Parienté, "Performance Analysis and Monitoring using Hardware Counters," *developers. sun. com/solaris/articles/hardware_counters. html*, 2001.

[104] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proceedings of the 3rd conference on Computing frontiers*. ACM, 2006, p. 40.

[105] M. Pettersson, "Linux x86 performance-monitoring counters driver," 2001.

[106] J. Marathe, A. Nagarajan, and F. Mueller, "Detailed cache coherence characterization for openmp benchmarks," in *ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing*. New York, NY, USA: ACM, 2004, pp. 287–297.

[107] J. Hourd, C. Fan, J. Zeng, Q. Zhang, M. Best, A. Fedorova, and C. Mustard, "Exploring Practical Benefits of Asymmetric Multicore Processors," *PESPMA 2009*, pp. 55–60, 2009.

[108] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1002/cpe.1631

[109] F. Bower, D. Sorin, and L. Cox, "The impact of dynamically heterogeneous multicore processors on thread scheduling," *IEEE Micro-Institute of Electrical and Electronics Engineers*, vol. 28, no. 3, pp. 17–25, 2008.

[110] K. Papadopoulos, K. Stavrou, and P. Trancoso, "Helpercoredb: Exploiting multicore technology to improve database performance," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–11.

[111] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys (CSUR)*, vol. 34, no. 2, pp. 171–210, 2002.