Scheduling local and remote memory in cluster computers

MÓNICA SERRANO GÓMEZ

EDITORIAL
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Universitat Politècnica de València

Departamento de Informática de Sistemas y Computadores

Scheduling Local and Remote Memory in Cluster Computers

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Engineering)

Author

Mónica Serrano Gómez

Advisors

Dr. Julio Sahuquillo Borrás
Dr. Houcine Hassan Mohamed

July 2013

Collection Doctoral Thesis

© Mónica Serrano Gómez

First Edition, 2014

© of the present edition: Editorial Universitat Politècnica de València
Telf.: 963 877 012 / www.lalibreria.upv.es

ISBN: 978-84-9048-219-3 (CD version)

Acknowledgements

This dissertation is mainly oriented to *big machines*, which already have the leading role through several sections of this document. Nevertheless, I can not put the work of *big people* to one side, so I would like to dedicate this modest part of the text to all of them. I am grateful to all those people who have shared their time and knowledge with me during all the steps of my research. On one hand, this thesis could not have been possible without the professional support of my Advisors, Julio Sahuquillo and Houcine Hassan, as well as the dedication of Salvador Petit, whose contributions to our meetings and technical advise have been essential to the developing of my work. Additionally, I show my gratitude to José Duato for its great work in our research group. On the other hand, I would like to highlight the human support, which has been as important as the technical support to get here. Working in a field in which most of the efforts are made to study performance indexes, a high *human performance* is very appreciated and encouraging. Thanks to my parents, colleagues, and friends to help my to grow as a better person while I was trying to do my best as a better researcher.

Abstract

Cluster computers represent a cost-effective alternative solution to supercomputers. In these systems, it is common to constrain the memory address space of a given processor to the local motherboard. Constraining the system in this way is much cheaper than using a full-fledged shared memory implementation among motherboards. However, memory usage among motherboards may be unfairly balanced depending on the memory requirements of the applications running on each motherboard. This situation can lead to disk-swapping, which severely degrades system performance, although there may be unused memory on other motherboards. A straightforward solution is to increase the amount of available memory in each motherboard, but the cost of this solution may become prohibitive.

On the other hand, remote memory access (RMA) hardware provides fast interconnects among the motherboards of a cluster computer. In recent works, this characteristic has been used to extend the addressable memory space of selected motherboards. In this work, the baseline machine uses this capability as a fast mechanism to allow the local OS to access to DRAM memory installed in a remote motherboard. In this context, efficient memory scheduling becomes a major concern since main memory latencies have a strong impact on the overall execution time of the applications, provided that remote memory accesses may be several orders of magnitude higher than local accesses. Additionally, changing the memory distribution is a slow process which may involve several motherboards, hence the memory scheduler needs to make sure that the target distribution provides better performance than the current one. This dissertation aims to address the aforementioned issues by proposing several memory scheduling policies.

First, an ideal algorithm and a heuristic strategy to assign main memory from the different memory regions are presented. Additionally, a Quality of Service control mechanism has been devised in order to prevent unacceptable performance degradation for the running applications. The ideal algorithm finds the optimal memory distribution but its computational cost is prohibitive for a high number of applications. This drawback is handled by the heuristic strategy, which approximates the best local and remote memory distribution among applications at an acceptable computational cost.

The previous algorithms are based on profiling. To deal with this potential shortcoming we focus on analytical solutions. This dissertation proposes an analytical model that estimates the execution time of a given application for a given memory distribution. This technique is used as a performance predictor that provides the input to a memory scheduler. The estimates are used by the memory scheduler to dynamically choose the optimal target memory distribution for each application running in the system in order to achieve the best overall performance.

Scheduling at a higher granularity allows simpler scheduler policies. This work studies the feasibility of scheduling at OS page granularity. A conventional hardware-based block interleaving and an OS-based page interleaving have been assumed as the baseline schemes. From the comparison of the two baseline schemes, we have concluded that only the performance of some applications is significantly affected by page-based interleaving. The reasons that cause this impact on performance have been studied and have provided the basis for the design of two OS-based memory allocation policies. The first one, namely on-demand (OD), is a simple strategy that works by placing new pages in local memory until this region is full, thus benefiting from the premise that most of the accessed pages are requested and allocated before than the least accessed ones to improve the performance. Nevertheless, in the absence of this premise for some benchmarks, OD performs worse. The second policy, namely Most-accessed in-local (Mail), is proposed to avoid this problem.

Resumen

Los clústers de computadores representan una solución alternativa a los supercomputadores. En este tipo de sistemas, se suele restringir el espacio de direccionamiento de memoria de un procesador dado a la placa madre local. Restringir el sistema de esta manera es mucho más barato que usar una implementación de memoria compartida entre las placas. Sin embargo, las diferentes necesidades de memoria de las aplicaciones que se ejecutan en cada placa pueden dar lugar a un desequilibrio en el uso de memoria entre las placas. Esta situación puede desencadenar intercambios de datos con el disco, los cuales degradan notablemente las prestaciones del sistema, a pesar de que pueda haber memoria no utilizada en otras placas. Una solución directa consiste en aumentar la cantidad de memoria disponible en cada placa, pero el coste de esta solución puede ser prohibitivo.

Por otra parte, el hardware de acceso a memoria remota (RMA) es una forma de facilitar interconexiones rápidas entre las placas de un clúster de computadores. En trabajos recientes, esta característica se ha usado para aumentar el espacio de direccionamiento en ciertas placas. En este trabajo, la máquina base usa esta capacidad como mecanismo rápido para permitir al sistema operativo local acceder a la memoria DRAM instalada en una placa remota. En este contexto, una planificación de memoria eficiente constituye una cuestión crítica, ya que las latencias de memoria tienen un impacto importante sobre el tiempo de ejecución global de las aplicaciones, debido a que las latencias de memoria remota pueden ser varios órdenes de magnitud más altas que los accesos locales. Además, el hecho de cambiar la distribución de memoria es un proceso lento que puede involucrar a varias placas, así pues, el planificador de memoria ha de asegurarse de que la distribución objetivo proporciona mejores prestaciones que la actual. La presente disertación pretende abordar los asuntos mencionados anteriormente mediante la propuesta de varias políticas de planificación de memoria.

En primer lugar, se presenta un algoritmo ideal y una estrategia heurística para asignar memoria principal ubicada en las diferentes regiones de memoria. Adicionalmente, se ha diseñado un mecanismo de control de Calidad de Servicio para evitar que las prestaciones de las aplicaciones en ejecución se degraden de forma inadmisible. El algoritmo ideal encuentra la distribución de memoria óptima pero su complejidad computacional es prohibitiva dado un alto número de aplicaciones. De este inconveniente se encarga la estrategia heurística, la cual se aproxima a la mejor distribución de memoria local y remota con un coste computacional aceptable.

Los algoritmos anteriores se basan en *profiling.* Para tratar este defecto potencial, nos centramos en soluciones analíticas. Esta disertación propone un modelo analítico que estima el tiempo de ejecución de una aplicación dada para cierta distribución de memoria. Dicha técnica se usa como un predictor de prestaciones que proporciona la información de entrada a un planificador de memoria. El planificador de memoria usa las estimaciones para elegir dinámicamente la distribución de memoria objetivo óptima para cada aplicación que se esté ejecutando en el sistema, de forma que se alcancen las mejores prestaciones globales.

La planificación a granularidad más alta permite políticas de planificación más simples. Este trabajo estudia la viabilidad de planificar a nivel de granularidad de página del sistema operativo. Un entrelazado convencional basado en hardware a nivel de bloque y un entrelazado a nivel de página de sistema operativo se han tomado como esquemas de referencia. De la comparación de ambos esquemas de referencia, hemos concluido que solo algunas aplicaciones se ven afectadas de forma significativa por el uso del entrelazado a nivel de página. Las razones que causan este impacto en las prestaciones han sido estudiadas y han definido la base para el diseño de dos políticas de distribución de memoria basadas en sistema operativo. La primera se denomina *on-demand* (OD), y es una estrategia simple que funciona colocando las páginas nuevas en memoria local hasta que dicha región se llena, de manera que se beneficia de la premisa de que las páginas más accedidas se piden y se ubican antes que las menos accedidas para mejorar las prestaciones. Sin embargo, ante la ausencia de dicha premisa para algunos de los benchmarks, OD funciona peor. La segunda política, denominada *Most-accessed in-local* (Mail), se propone con el objetivo de evitar este problema.

Resum

Els clústers d'ordinadors representen una solució alternativa als superordinadors. En aquest tipus de sistemes, se sol restringir l'espai d'adreçament de memòria d'un processador donat a la placa mare local. Restringir el sistema d'aquesta manera és molt més barat que usar una implementació de memòria compartida entre les plaques. No obstant això, les diferents necessitats de memòria de les aplicacions que s'executen en cada placa poden donar lloc a un desequilibri en l'ús de memòria entre les plaques. Aquesta situació pot desencadenar intercanvis de dades amb el disc, els quals degraden notablement les prestacions del sistema, tot i que puga haver-hi memòria no utilitzada en altres plaques. Una solució directa consisteix a augmentar la quantitat de memòria disponible en cada placa, però el cost d'aquesta solució pot ser prohibitiu.

D'altra banda, el hardware d'accés a memòria remota (RMA) és una forma de facilitar interconnexions ràpides entre les plaques d'un clúster de computadors. En treballs recents, aquesta característica s'ha usat per a augmentar l'espai d'adreçament en certes plaques. En aquest treball, la màquina base fa servir aquesta capacitat com a mecanisme ràpid per a permetre al sistema operatiu local accedir a la memòria DRAM instal·lada en una placa remota. En aquest context, una planificació de memòria eficient constitueix una qüestió crítica, ja que les latències de memòria tenen un impacte important sobre el temps d'execució global de les aplicacions, pel fet que les latències de memòria remota poden ser diversos ordres de magnitud més altes que els accessos locals. A més, el fet de canviar la distribució de memòria és un procés lent que pot involucrar a diverses plaques, així doncs, el planificador de memòria ha d'assegurar-se que la distribució objectiu proporciona millors prestacions que l'actual. La present dissertació pretén abordar els assumptes esmentats anteriorment mitjançant la proposta de diverses polítiques de planificació de memòria.

En primer lloc, es presenta un algorisme ideal i una estratègia heurística per a assignar memòria principal situada en les diferents regions de memòria. Addicionalment, s'ha dissenyat un mecanisme de control de Qualitat de Servei per tal d'evitar que les prestacions de les aplicacions en execució es degraden de forma inadmissible. L'algorisme ideal troba la distribució de memòria òptima però la seua complexitat computacional és prohibitiva donat un alt nombre d'aplicacions. D'aquest inconvenient s'encarrega

l'estratègia heurística, la qual s'aproxima a la millor distribució de memòria local i remota amb un cost computacional acceptable.

Els algorismes anteriors es basen en *profiling*. Per tractar aquest defecte potencial, ens centrem en solucions analítiques. Aquesta dissertació proposa un model analític que estima el temps d'execució d'una aplicació donada per a certa distribució de memòria. Aquesta tècnica s'usa com un predictor de prestacions que proporciona la informació d'entrada a un planificador de memòria. El planificador de memòria usa les estimacions per a triar dinàmicament la distribució de memòria objectiu òptima per a cadascuna de les aplicacions que s'estan executant en el sistema, de manera que s'aconseguisquen les millors prestacions globals.

La planificació a granularitat més alta permet polítiques de planificació més simples. Aquest treball estudia la viabilitat de planificar a nivell de granularitat de pàgina del sistema operatiu. Un entrellaçat convencional basat en hardware a nivell de bloc i un entrellaçat a nivell de pàgina de sistema operatiu s'han pres com a esquemes de referència. De la comparació de tots dos esquemes de referència, hem conclòs que solament algunes aplicacions es veuen afectades de forma significativa per l'ús de l'entrellaçat a nivell de pàgina. Les raons que causen aquest impacte en les prestacions han estat estudiades i han definit la base per al disseny de dues polítiques de distribució de memòria basades en sistema operatiu. La primera es denomina *on-demand* (OD), i és una estratègia simple que funciona col·locant les pàgines noves en memòria local fins que aquesta regió s'omple, de manera que es beneficia de la premissa que les pàgines més accedides es demanen i se situen abans que les menys accedides per tal de millorar les prestacions. No obstant això, davant l'absència d'aquesta premissa per a alguns dels benchmarks, OD funciona pitjor. La segona política, denominada *Most-accessed in-local* (Mail), es proposa amb l'objectiu d'evitar aquest problema.

Contents

List of Figures

List of Tables

Abbreviations and Acronyms

AMD	**A**dvanced **M**icro **D**evices
ATM	**A**synchronous **T**ransfer **M**ode
BI	**B**lock-level **I**nterleaved
CAMP	**C**ache **A**ware performance model for **M**ulti-core **P**rocessors
CMP	**C**hip **M**ulti**P**rocessors
CPU	**C**entral **P**rocessing **U**nit
DI	**D**istributed **I**ntensity
DIO	**D**istributed **I**ntensity **O**nline
DLM	**D**istributed **L**arge **M**emory
DSM	**D**istributed **S**hared **M**emory
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
ELF	**E**xecutable and **L**inkable **F**ormat
GPU	**G**raphics **P**rocessing **U**nit
IPC	**I**nstructions **P**er **C**ycle
IQ	**I**nstruction **Q**ueue
ISA	**I**nstruction **S**et **A**rchitecture
L	**L**ocal memory
LA	**L**east **A**ccessed pages
Lb	**L**ocal to **b**oard memory
LRU	**L**east **R**ecently **U**sed
LSQ	**L**oad and **S**tore **Q**ueue
MA	**M**ost **A**ccessed pages
Mail	**M**ost **A**ccessed **in** **l**ocal

MICEMemO	**M**odel and **I**terative **C**ompilation for **E**ffective **Mem**ory **O**ptimization
MM	**M**ain **M**emory
MPKI	**M**isses **P**er **K**ilo-**I**nstruction
MSHR	**M**iss **S**tatus **H**old **R**egister
NUMA	**N**on-**U**niform **M**emory **A**ccess
OD	**O**n **D**emand
OS	**O**perting **S**ystem
PC	**P**ersonal **C**omputer
PI	**P**age-level **I**nterleaved
PNR	**P**arallel **N**etwork **R**AM
QoS	**Q**uality **o**f **S**ervice
R	**R**emote memory
RMA	**R**emote **M**emory **A**ccess
SMP	**S**ymmetric (shared-Memory) **M**ulti**P**rocessor
SPP	**S**et of **P**ossible **P**ermutations
TLB	**T**ranslation **L**ookaside **B**uffer
VLSI	**V**ery-**L**arge-**S**cale **I**ntegration

Chapter 1

Introduction

This chapter introduces some concepts and presents the motivation for the work developed in this thesis. First, some basic notions about parallel computers are given in order to introduce the system to which this work is targeted. Then, it is shown how the described system is affected by the problem of memory unbalance and the high latencies required to access certain memory regions. Finally, it is summarized how the rest of this dissertation deals with this problem through different memory scheduling proposals.

1.1 Background on Parallel Computing Architectures

For over a decade, the growth in performance and capability of computer systems has been explosive, mainly due to the advance of the underlying VLSI technology, which allows larger and larger numbers of components to fit on a chip and clock rates to increase. This fact translates the raw potential of the technology into greater performance and expanded capability of the computer system. This change has been mainly achieved thanks to parallelism. A larger volume of resources means that more operations can be done at once, in parallel. Parallel computer architecture is about organizing these resources so that they work well together. All kind of systems have implemented parallelism more and more effectively to gain performance, and the level at which parallelism is exploited continues to rise. The other key character is storage. The data that is operated at an ever faster rate must be held somewhere in the machine. Thus, parallel processing is deeply related with data locality and communication. These changing relationships are the main concerns to design the various levels of a computer system so as to maximize performance and programmability within the limits imposed by technology and cost at any particular time.

Parallelism is an interesting perspective from which to understand computer architecture because it applies at all levels of design, it interacts with essentially all other architectural concepts, and it presents a unique dependence on the underlying technology. In particular, the basic issues of locality, bandwidth, latency, and synchronization arise at many levels of the design of parallel computer systems. The trade-offs must be resolved in the context of real application workloads.

There is a wide range of architectural styles of parallel machines [2]. Throughout the following sections we will only describe those which are essential to the understanding of the work discussed in this dissertation.

1.1.1 Message-passing Communication Model

The demand for even more computer power to deal with high-performance computing has been continuously increasing during the last decades, and message-passing based systems have been one of the commonly used approaches.

The address space consists of multiple private address spaces that are logically disjoint and cannot be addressed by a remote processor. In such multiprocessors, the same physical address on two different processors refers to two different locations in two different memories. Each processor-memory pair is essentially a separate computer. Initially, such computers were built with different processing nodes and specialized interconnection networks. In such a multiprocessor system (with multiple address spaces), communication of data is done by explicitly passing messages among the processors. Therefore, these multiprocessors are called message-passing multiprocessors.

This approach is highly scalable with the number of processors (e.g., BlueGene/P can be scaled up to 884,736-processors [1]). A major drawback of the message-passing model is that it complicates the programming of parallel applications.

1.1.2 Shared Memory Architectures

To overcome the drawbacks of message-passing model, the industry has moved to shared memory systems for small to medium number of processors.

Shared memory architectures constitute one of the most important classes of parallel machines. The key property of this class is that communication occurs implicitly as a result of conventional memory access instructions, i.e., loads and stores. This kind of architectures have existed since the early 60s, and still today they have a role in almost every segment of the computer industry. Shared memory multiprocessors serve to provide better throughput on multiprogrammed workloads, as well as to support parallel programs. Thus, they are naturally found across a wide scaling range, from a few processors to perhaps hundreds.

This kind of systems can be classified in two main categories. In the first category, *Symmetric Shared-Memory Multiprocessors* (SMP), multiple processor-cache subsystems share the same physical memory, typically connected by one or more buses or a switch. The key architectural property is the uniform access time to all the main memory from all the processors. Figure 1.1 shows a block diagram of a typical SMP. This type of symmetric shared-memory architecture is currently by far the most popular organization [1].

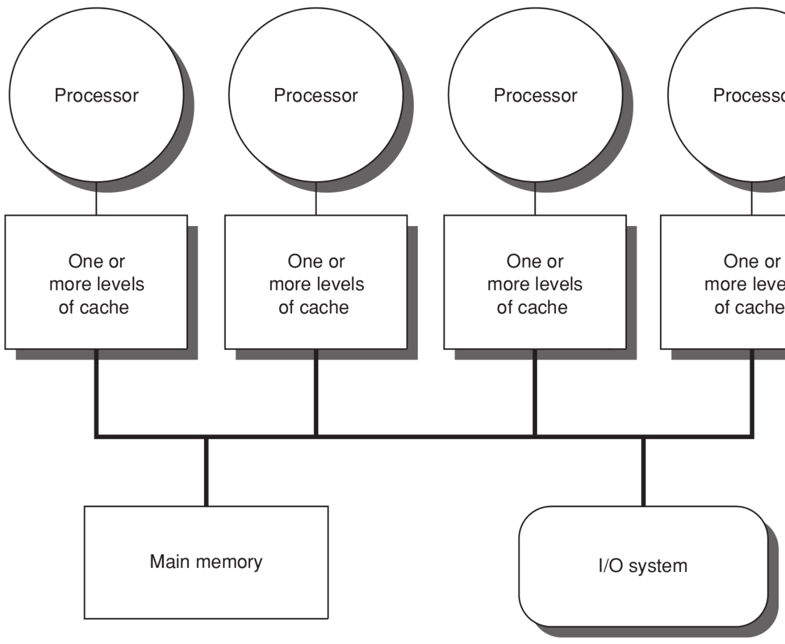

Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
I/O system

FIGURE 1.1: Basic structure of a centralized SMP. Source [1].

These systems are relatively expensive and they do not scale to large sizes (e.g., larger than 16 nodes) since they use a common shared bus to access to main memory. Projects working on shared memory with coherent cache, like the NUMAChip by Dolphin Interconnect Solutions [3], suffer from limited scalability introduced by the coherence protocol. Thus, a major concern is that the access to remote memory becomes affordable and efficient both regarding to latency and price.

To support larger processor counts, memory must be distributed among the processors rather than centralized; otherwise the memory system would not be able to support the bandwidth demands of a larger number of processors without incurring excessively long access latency. This fact justifies the arisen of the second category of Shared Memory Architectures: *Distributed Shared Memory* (DSM) [4]. DSMs provide a virtual address space shared among processes running on loosely coupled processors. The physically separate memories can be addressed as one logically shared address space, meaning that a memory reference can be made by any processor to any memory location, assuming that it has the correct access rights. Figure 1.2 shows what these multiprocessors look like. With the rapid increase in processor performance and the associated increase in the memory bandwidth requirements, the size of a multiprocessor for which distributed memory is preferred continues to shrink.

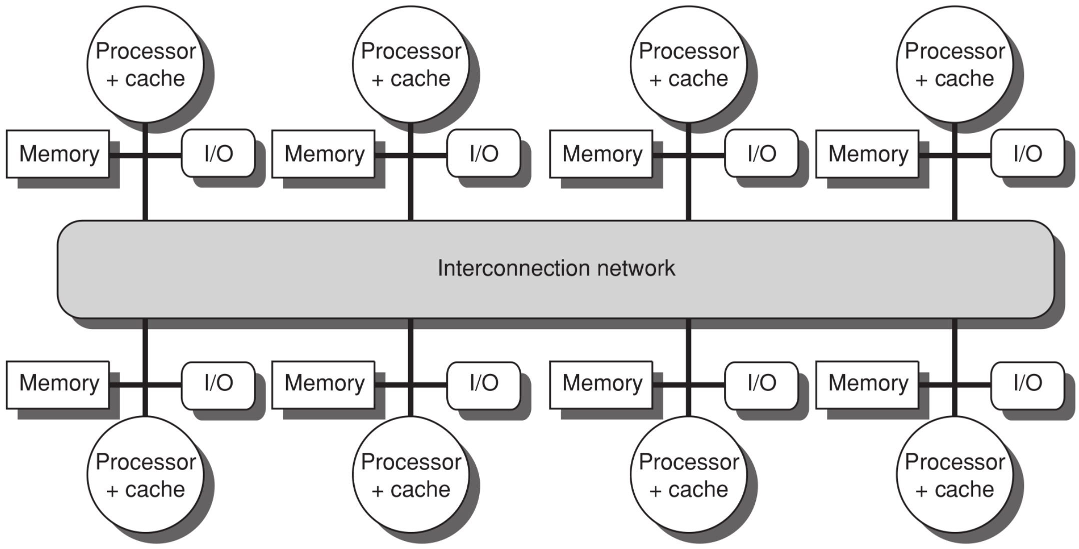


Figure 1.2: Basic architecture of a DSM. Source [1].

Distributing the memory among the nodes has two major benefits. First, it is a cost-effective way to scale the memory bandwidth if most of the accesses are performed to the local memory in the node. Second, it reduces the latency of the accesses to the local memory. These two advantages make distributed memory attractive at smaller processor counts as processors get even faster and require more memory bandwidth and lower memory latency. The key disadvantages for a distributed-memory architecture are that communicating data between processors becomes somewhat more complex, and that it requires more software effort to take advantage of the increased memory bandwidth supplied by distributed memories.

1.1.3 Multicore Processors

On the other hand, technology constraints have moved chip manufacturers from complex cores to simpler multicore based processors [5]. Starting in the 1990s, the increasing capacity of a single chip allowed designers to place multiple processors on a single die. In such a design, the multiple cores typically share some resources, such as a second or third-level cache or memory and I/O buses. Recent processors, including the IBM Power5, the Sun T1, and the Intel Pentium D and Xeon-MP, are multicore and multithreaded. Just as using multiple copies of a microprocessor in a multiprocessor leverages a design investment through replication, a multicore achieves the same advantage by relying more on replication than the alternative of building a wider superscalar.

1.1.4 Cluster Computers

Cluster computers present an alternative solution to supercomputers offering a good tradeoff between price and performance. These systems currently constitute an important segment of the market since they can use free software and provide good performance for a wide diversity of applications such as high-performance parallel computing, e-business, or user applications running concurrently.

These systems consist of a set of interconnected motherboards. Each motherboard can be seen as a block of a cluster and hosts a number of processors (usually multicore), which is expected to dramatically grow with future technologies.

Their affordable price and their potential computational power has led clusters to grow in popularity to the detriment of conventional supercomputers. Nevertheless, the shared memory space that can be seen by a processor is limited to the available memory within the motherboard. We refer to this kind of machine as a cluster of DSMs or the *original machine*.

This work is a part of a wider research project working on a real cluster of DSMs where the main goal is to devise new memory scheduling algorithms targeted to this machine.

1.2 Motivation

Large scientific parallel applications and a wide set of commercial applications (e.g., query processing databases) demand large amounts of memory space. Current parallel computing platforms schedule jobs without fully knowing their memory requirements. This leads to unbalanced memory allocation in which some nodes are overloaded. Consequently, the contents of memory must be swapped out to a storage device in those motherboards whose memory requirements exceed their memory capacity. Thus, system performance is severely degraded while the memory in other motherboards may be underused.

A straightforward solution to address the previous shortcoming is to oversize DRAM memory in the nodes; however, this solution may become prohibitive as this device is one of the most expensive resources on high performance computers. IBM z series [6] and HP Integrity Superdome [7] mainframes are examples of shared-memory machines with an amount of expensive memory that can be as large as two Terabytes.

On the contrary, the solution proposed in this work is able to perform without extra resources because it takes advantage of free or unused memory on a remote motherboard of the system in order to increase the available memory for a local application. In fact, our approach may even do better than oversizing, both in terms of performance and hardware cost. See the example below:

> Take a node with an amount of memory X, in which an application requiring an amount Z, $X < Z$, is running. Let's analyze the implications which would have each of the two choices above. *i*) Increasing the node's

local memory up to Y, being $Y < Z$, would imply to borrow the memory exceeding Y from hard disk.
ii) On the other hand, just taking $Z - X$ from another board (which does not need it) would avoid accessing to the hard disk, whose latency is several orders of magnitude greater than accessing to main memory.

The latter choice (*ii*) is a cheap solution compared to upgrading the installed memory, as it saves the cost of adding memory to all the motherboards. In addition, the fact of avoiding disk swapping will lead to better performance.

This dissertation assumes that the target system is provided with two main capabilities. First, the local OS is able to see the remote memory installed in other motherboards. Second, the access to remote memory is performed through a fast interconnection mechanism. For this purpose, Remote Memory Access (RMA) hardware[8] is installed in the original machine. RMA mechanisms allow the access to memory in remote motherboards with reasonable latencies [9], that is, like in shared memory systems, the application address space is allowed to span beyond a motherboard. Therefore, the resulting system will require an OS-level allocation protocol to reserve remote segments of memory. Following this protocol, an operating system running on a given motherboard will ensure that each allocated segment of remote memory can be only accessed by one motherboard.

High-end systems like BlueGene/L [10], BlueGene/P [11], Cray XT, etc. [12] include RMA hardware as a mechanism to reduce the communication latency even when using the message passing programming model. Although RMA is currently only a feature of modern high-end systems, it is expected to find commodity implementations in the near future [9].

With regard to the memory coherence issue in the system, most applications take advantage of having more memory resources but do not need additional computing nodes other than those included in the local board. So, as they do not use processors in the remote board, they can perform without memory coherence [13, 14] in that board.

In other words, coherence is activated within the board (i.e., intra-motherboard) but it is not among different motherboards (i.e., inter-motherboards).

We will refer to such a system as the *baseline* machine. The research on this manuscript is focused on scheduling policies to efficiently handle local and remote memory on the *baseline* machine.

1.3 Objectives of the Thesis

The execution time of a huge amount of current applications mainly depends on how efficiently the system handles the memory accesses. In the target system, memory management is a critical issue for performance since the main memory used by the applications can reside in three main locations which have widely different memory latencies, i) in the same node as the processor running the application (*Local to Node* or L), ii) in other node of the local motherboard (*Local to Board* or Lb), and iii) in a remote motherboard (*Remote* or R). Notice that in this scenario the additional memory availability comes at expense of much longer latencies, thus an efficient memory scheduling is required to improve the performance of a given set of applications.

The main objective of this dissertation is to devise efficient memory scheduling algorithms that assign memory (local and remote) to applications in order to provide the best system performance, while guaranteeing a minimum quality of service QoS to each application.

This general objective can be in turn divided in sub-objectives. First, the behavior of the different applications must be characterized from the main memory perspective, taking into account the number of accessed pages, how likely each page is accessed, the working set size, etc. Second, based on this characterization this dissertation pursues to design memory schedulers to deal with performance. Two different types of algorithms will be devised. On one hand, the algorithms based on information taken from a previous profile and, on the other hand, the policies that use run-time information.

In addition, the impact of considering different memory-item granularities (e.g. block size or page size) from the scheduling point of view will be analyzed.

1.4 Contributions of the Thesis

In this dissertation, several memory scheduling policies have been proposed to efficiently handle the memory used by the applications running in cluster computers. The main memory of these systems is assumed to be divided in three main locations (i.e., L, Lb, R) whose access latencies widely differ. In this scenario, all the proposals are focused on improving the overall performance of the system. The contributions of this thesis can be summarized as follows:

- An Ideal algorithm and a simple and Cost-effective Heuristic are devised to statically schedule the local and remote memory of the target machine while taking into account quality of service(QoS) constraints.
- A Performance Predictor that allows the memory scheduler to dynamically choose the optimal memory distribution is implemented.
- Two memory scheduling policies, namely on-demand (OD) and Most-accessed in-local (*Mail*), which work at page granularity are devised.

Regarding to the first major contribution, as a first step we study the impact of the accesses to the different memory regions in the system performance for different types of workload. Results show important performance drops when a given application accesses the remote memory region. A wide variability in the impact on performance is exhibited among the different applications analyzed in the experiments. From the results of this analysis, an Ideal memory scheduling algorithm (namely SPP) is designed. SPP is fed by an off-line profile of the benchmarks for different memory distributions, from which it schedules memory among applications, while guaranteeing a minimum QoS to each application. The implementation of the ideal SPP is infeasible due to its cost. To deal with this drawback we devise a Cost-effective memory scheduling

Heuristic. In both cases, we assume that the distribution of local and remote memory assigned to an application is set statically by interleaving memory addresses at cache block size level (64B).

With respect to the second major contribution, the proposed predictor dynamically estimates the performance of the benchmarks by measuring their utilization of the system resources during the execution. The devised Performance Predictor is driven by a novel analytical model fed by simple hardware counters, available in most current processors, which gather the amount of microprocessor cycles spent in computation, memory access, and network resources usage. The estimates of the Performance Predictor constitute the input to the memory scheduler, which uses this information to choose at run-time the optimal (from the system performance point of view) target memory distribution for each application running in the system.

Finally, the operating system (i.e., the scheduler) manages memory at page level granularity. Thus, to ease the scheduler job, we investigated the feasibility of supporting interleaved memory at OS page granularity. Results show that this solution does not impact on the performance of most of the benchmarks. Based on this observation we looked for the reasons of performance drops in those benchmarks showing worse performance when working at page granularity. The results of this analysis led us to propose two memory allocation policies. The OD policy first places the requested pages in local memory, once this memory region is full, the subsequent memory pages are placed in remote memory. However, OD shows some performance drawbacks, which are solved by the Mail allocation policy.

1.5 Thesis Outline

The rest of this dissertation is structured as follows. Chapter 2 includes some related work dealing with three main issues: remote memory mainly used as a swapping area to avoid access to disk, models to estimate the memory system performance, and memory system aware scheduling techniques. Chapter 3 presents the system prototype and the model of the system, the simulation framework and the workloads selected for the experiments. Chapter 4 describes and evaluates a simple static scheduling policy referred to as Cost-effective Heuristic. Chapter 5 presents a Performance Predictor which enhances the scheduling algorithm presented in Chapter 4 by dynamically estimating the performance for a given memory distribution. Chapter 6 introduces scheduling policies based on page granularity. Finally, Chapter 7 presents some concluding remarks.

Chapter 2

Related Work

In this chapter, some work related with this thesis is discussed. The cited approaches are classified in three sections depending on whether they propose the use remote memory as an improvement of disk swapping, address the problem of estimating the memory system performance by means of different performance models, or deal with the scheduling of memory resources to mitigate latencies and thus enhancing the performance.

2.1 Introduction

Many works have been performed considering local and remote memory, but their focus differs from the considered in this work. In this chapter we classify this work in three main categories according to the specific problem of the system that they address.

Papers in the first category mainly focus on the use of remote memory as an alternative and relatively fast (compared to hard disks) memory device for swapping purposes, instead of using remote memory to extend the local main memory address space.

The second category groups papers addressing the problem of estimating performance when accessing shared memory structures, mainly concentrating on memory contention. Nevertheless, none of the proposed performance models considers remote memory access.

Finally, the latter category briefly describes a set of papers dealing with the usage of scheduling mechanisms which intend to reduce contention points when accessing to shared resources to enhance performance.

2.2 Proposals Focusing on the Use of Remote Memory for Swapping

Different research papers dealing with remote memory allocation and mostly related to memory swap can be found in the literature. The referred papers use remote memory for swapping over cluster nodes and present their approaches as an improvement of disk swapping.

In [15] authors develop a software-based prototype by extending the Xen hypervisor to emulate a disaggregated memory design wherein remote pages are swapped into local memory on-demand upon access. This design is presented as a cost-effective way to scale memory capacity. Their results reveal that low-latency remote memory calls for a different regime of replacement policies than conventional disk paging and show the synergy between disaggregated memory and content-based page sharing. They find that a combination of remote and local memory distribution provides higher workload consolidation opportunity and performance-per-cost than either technique alone. Their study also shows that disaggregated memory provides similar response time performance at a lower cost compared to scaling out on multiple compute blades, thus demonstrating the feasibility of the software infrastructure required for disaggregated memory.

Midorikawa et al. propose the distributed large memory system (DLM), which is a user-level software-only solution that provides very large virtual memory by using remote memory distributed over the nodes in a cluster [16]. The performance of DLM programs that access remote memory is compared to ordinary programs that use local memory. The results of STREAM, NPB and Himeno benchmarks show that the DLM achieves better performance than other remote paging schemes using a block swap device to access remote memory. To obtain high performance, the DLM can tune its parameters independently from kernel swap parameters. In addition to performance, DLM offers the advantages of easy availability and high portability, because it does not need special hardware.

Another work of Midorikawa [17] proposes a page size control methodology that estimates a working data set and changes page size to each processing part of an application when running to prevent memory server thrashing. Adaptive page size is performed by unified transmission of multiple basic minimum pages. It also supports a transmission of a transient fragmented large page generated when page size is changed from small to large. Users can set favorite basic minimum page size and initial start page size when they run their programs if they do not want to use the default values. This is a simple and effective methodology that is applicable to various page-based memory accessing systems, like distributed shared memory and general paging systems, especially for applications with various memory access patterns.

Shuang et al. design a remote paging system for remote memory utilization in InfiniBand clusters [18]. They aim to benefit from the low latency and high bandwidth of Infiniband networks to reduce the latency gap between access to local memory and remote memory in modern clusters. Remote idle memory is presented as a resource that can be exploited to reduce the memory pressure on individual nodes. They explain that the fact of adding an additional level in the memory hierarchy between local memory and the disk leads to dramatic performance improvements specially for memory intensive applications. Their work presents the design and implementation of a high performance networking block device over InfiniBand fabric, which serves as a swap device of a virtual memory system for efficient page transfer to/from remote memory servers. They demonstrate that, under their implementation, quicksort performs 1.45 times slower than local memory system, and up to 21 times faster than local disk. Finally, they identify that the host overhead is a key issue for further performance improvement of remote paging over high performance interconnects clusters.

In [19], the use of remote memory for virtual memory swapping in a cluster computer is described. The system, which is called LocaSwap, uses a lightweight kernel-to-kernel communications channel for fast and efficient data transfer. It utilizes an Ethernet network to interconnect PCs into a cluster. Performance tests are made to compare the proposed system to normal hard disk swapping. Performance results show considerable improvement over the use of hard disks. In particular, the random read performance is significantly better with local swap. Finally, given a fixed number of reads, LocaSwap's time is only slightly affected by the size of remote memory while hard disk performance degrades linearly as the size of the swap space increases.

Oleszkiewicz et al. propose a peer-to-peer solution called Parallel Network RAM (PNR) [20] which allows parallel jobs to utilize idle remote memory. In this scheme, each node requests memory resources and provides memory for other nodes through a local manager (super-peer). This manager is in charge of coordinating the allocation of network RAM of several nodes and ensuring that load is evenly distributed to the nodes hosting parallel processes belonging to the same parallel job. PNR allows more jobs to execute concurrently without resorting to disk paging and it makes a more efficient use of the available RAM resources in a cluster, especially in clusters with unbalanced resource utilization. It reduces the computational, communication and synchronization overhead typically involved in parallel applications. This leads to decrease average response time and to achieve higher system throughput.

Jeon et al. present a user-level remote memory system [21] that processes large graph data when main memory space is insufficient to store application data. They exploit the efficient low-latency, high-performance feature of InfiniBand networks as well as the use of Remote Direct Memory Access operations to reduce the access time gap/bandwidth between local main memory and remote memory. In this way, the proposed remote memory design not only improves disk-paging systems but also achieves performance results comparable to that of main memory without requiring any special algorithm for remote memory. In this paper, authors also present their implementation based on remote memory to deal with large data sets.

In [22] Krishnan et al. describe and evaluate the scalability of linear algebra kernels based on a remote memory access approach. They discuss the performance and scalability of two popular parallel linear algebra kernels – matrix multiplication and LU factorization. Their design is targeted to an architectural model based on a cluster of multiprocessor nodes with a network that supports remote memory access (RMA) communication between nodes. Experimental results using large scale systems (Linux-InfiniBand cluster, Cray XT) demonstrate consistent performance advantages over ScaLAPACK suite, the leading implementation of parallel linear algebra algorithms used today.

Oguchi et al. [23] explain a method by means of which nodes executing applications dynamically acquire extra memory from remote nodes through an ATM network. The idle nodes are statically selected and called *memory servers*. When the amount of memory used in the local node exceeds the value of a parameter that limits the memory usage, part of the memory contents are swapped out (following a LRU algorithm) to the available memory in remote idle nodes. Each time the local node tries to access an item that has been swapped out, a page fault occurs. Then it calculates by means of a hash function which memory server has to send the requested item back. This technique is considerably better than using a hard disk as a swapping device. However, using dynamic remote memory acquisition with simple swapping leads to a high number of page faults when the memory usage limit parameter is small. To address this drawback, authors propose another dynamic acquisition method with remote memory update operations which restricts memory swapping and achieves better performance.

Also in the context of parallel data mining in ATM-connected PC clusters, Oguchi et al. [24] investigate the feasibility of using the available memory of remote nodes as a swap area when some nodes need to swap out their real memory contents. They analyze the association rule mining problem, which has a peculiar use of main memory as it allocates many small data areas in main memory accessed almost at random. The number of those areas multiplies to be enormous during the execution, leading to a dynamically changing requirement of memory space and swapping out to a secondary storage system. Consequently, the performance of the system severely degrades. In this sense, a method of remote memory utilization with update operations which improves the use of a hard disk as a swapping device is proposed and extended with a dynamic decision mechanism for remote memory availability. Finally, the migration process is evaluated, concluding that the overhead of memory contents migration is almost negligible unless the interval of monitoring the amount of available memory is too short.

2.3 Proposals Focusing on Estimating Memory System Performance

Some research papers can be found in the bibliography focusing on performance models mainly constrained by the memory system. Unlike this thesis, no remote memory is considered and most of them concentrate on estimating contention when accessing shared memory structures (e.g. caches or local memory).

Pingjing et al. [25] propose the use of memory optimization methods to alleviate the impact of the memory wall on performance of the programs. There are two main kinds of optimization methods to compute optimal optimization parameters: static or model-driven approaches, and empirical or execution-driven methods. The latter is more effective but quite time consuming. For this reason, authors devise a combination of model driven and empirical optimization methods: Combining Model and Iterative Compilation for Effective Memory Optimization (MICEMemO). This approach utilizes apriori information from hardware performance counters collected from a few runs of the program to narrow the optimization space, and then uses genetic algorithms to select good optimization parameters. Experimental results demonstrate that MICEMemO can greatly reduce memory access time, and the influence ratio for memory reference.

To determine the behavior of several applications sharing cache memory a multicore processor, Xie. et al. [26] propose an animal-based classification algorithm which can accurately predict when cache sharing interference problems may arise and consequently apply dynamic cache partitioning techniques. They implement the solution on hardware to allow dynamic classification of applications behaviors. Their proposal consists of a simple dynamic cache partitioning policy performing slightly better while incurring a lower implementation cost than the Utility-based Cache Partitioning scheme.

Xu et al. [27] propose CAMP, a fast and accurate shared cache aware performance model that estimates the performance degradation due to cache contention of processes running on CMPs. They use non-linear equilibrium equations in a least-recently-used (LRU) or pseudo-LRU last-level cache, taking into account process reuse distance histograms, cache access frequencies, and cache miss rate of each process to predict its effective cache size when sharing cache with other processes, allowing instruction throughput estimation. They also propose an easy-to-use method of obtaining the reuse distance histograms of a process that uses only commonly available hardware performance counters, without offline simulation or modification to commodity hardware or operating system. CAMP achieves an average performance prediction error of 1.57%.

In [28] the authors apply machine learning techniques to predict the performance on multicore processors with reasonable accuracy. These techniques do not require specialized hardware support and can reduce the time devoted to performance prediction, thus amortizing the time investment in training the algorithm to build the model. In their study, they show that a number of key solo-run program attributes can be used to predict paired-run performance. The paired run involves the contention for shared resources between co-running programs, mainly focusing on L2 caches.

2.4 Proposals on Memory System Aware Scheduling

As proposed in this work, scheduling resources can help mitigate latencies. Many research work focusing on scheduling has concentrated on reducing contention points of the system to enhance the performance. These points appear when accessing to shared resources (e.g. caches, main memory controller, main memory modules, etc).

A representative work on these topics is the paper by Zhuravlev et. al [29]. This work presents a comprehensive analysis of contention-mitigating techniques and identify the contention points that impact on performance degradation. The study is experimentally performed in an Intel Xeon X3565 quad-core processor. To mitigate performance degradation due to contention points they propose two scheduling algorithms DI and DIO that distribute threads such that the miss rate is evenly distributed among caches, with the aim of minimizing the cache miss rate. They conclude that the highest impact on performance is on improving quality of service or performance isolation for individual applications, and not on improving performance of a workload as a whole.

An interesting work is the paper by Dong et al. [30] that proposes a 3D system with different memory latencies (on-chip and off-chip). They address the problem at the memory controller side, which can be pure hardware-based or OS-assisted, depending on the migration granularity. The memory controller includes a component to act as a scheduler, and is able to decide which information should be stored on-chip and which one in the off-chip memory.

Antonopoulos et al. [31, 32] make a wide research on bandwidth-aware multicore scheduling to mitigate the performance penalties due to memory contention. In this sense, they propose several scheduling policies based on the memory bus bandwidth consumption of the processes running at the same time (from now on co-runners).

In [31], the bus bandwidth consumption values are obtained by modifying the source code of the running applications, while in [32], less intrusive implementations based on processor performance information are explored. In both cases, the proposed policies try to match the total bandwidth requirements of the co-runners to the peak memory bus bandwidth. In a posterior work addressing SMP clusters [33], Koukis et al. take into account the network bandwidth as well.

Finally, the novelty in the work of Nikolopoulos [34] is the fact that it deals with the remote memory access issue. This paper presents a methodology for quantifying remote memory access contention on hardware cache-coherent DSM multiprocessors. To this end, the number of accesses from each node to each page in memory is collected in hardware page reference counters (available in many commercial DSM systems) during the execution of the program. From this information, the methodology estimates the fraction of execution time wasted in contention. Additionally, an algorithm which balances the remote memory accesses across the DSM nodes to reduce the execution time of parallel applications is proposed. The proposed technique consists in detecting potential hot spots in pages and resolving contention on them using dynamic page migration. This algorithm is evaluated on a 128-processor Origin2000, proving that it is able to alleviate contention and reduce the parallel execution time of six application benchmarks by 19–34%.

2.5 Summary

In this chapter, some previous work dealing with the memory handling issue in cluster computers has been presented. They range from the use of remote memory as an alternative to disk swapping, to the addressing of the memory-aware performance estimation problem, as well as the usage of scheduling techniques to improve the system performance.

The novelties of the work proposed in this thesis with respect to the cited works reside in three key aspects: i) the use of remote memory not as a swapping device but as an extension of the main memory installed in a given local motherboard, which can be seen by the OS running in this local board; ii) the focus on both local and remote memory to devise a performance estimation model of the system (beyond the cache level or just the local memory, as other authors do), and iii) a memory scheduler which distributes local and remote memory among the running applications depending on their memory requirements while guaranteeing the requested QoS for all of them.

Chapter 3

System Model and Experimental Framework

As a preface to the main proposals devised in this dissertation, the hardware and software framework in which the work is conceived are presented in this chapter.

Our research concentrates on the domain of cluster computers, specifically focused in an efficient management of their memory resources to improve the system performance. The target system consists of a high performance cluster machine which has been equipped with specific hardware capabilities as well as OS memory allocation protocols to span main memory storage beyond the local motherboard.

The system was modeled on a extended version of the Multi2Sim simulation framework, and several benchmarks suites were selected to carry out the experiments. Both the simulation tool and the workloads are described in detail through this chapter.

3.1 Target System

3.1.1 Cluster

This work was originally initiated with the aim of enhancing main memory management in a cluster-based machine prototype. A cluster machine with the required hardware/software capabilities was being prototyped in conjunction with researchers from the University of Heidelberg [9], which designed the RMA connection cards. The machine (see Figure 3.1) consists of 64 motherboards, each one including four quad-core 2.0GHz Opteron processors in a 4-node NUMA system (one processor per node), and an amount of 16GB RAM memory per motherboard. This work is mainly targeted at analyzing and scheduling the main memory physically distributed among the OS (i.e., processes) running in different motherboards. We assume that extra memory is available in a remote motherboard and focus on scheduling strategies to enhance the system performance.

In this cluster machine, an application can both use its local memory and allocate memory in remote motherboards.

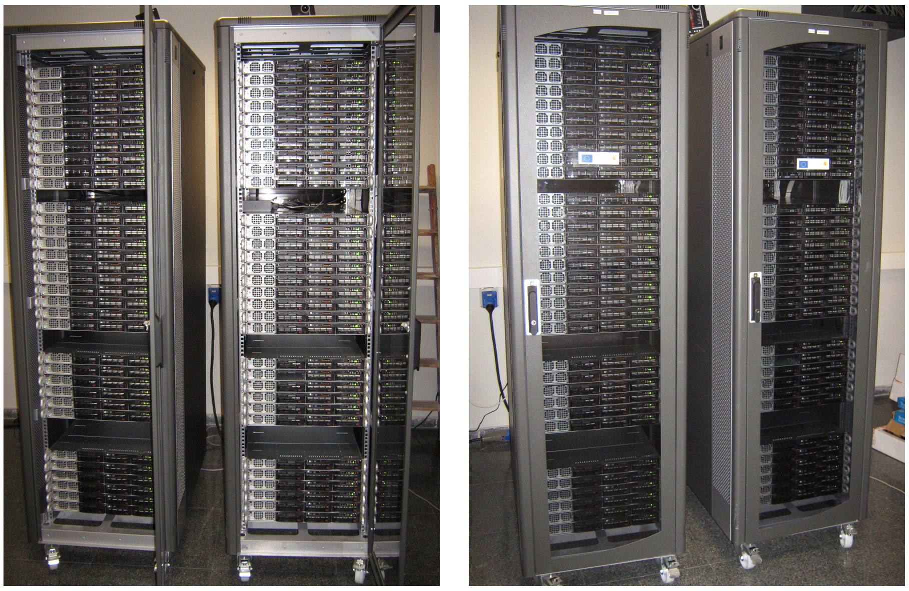


Figure 3.1: Prototype pictures.

3.1.2 Standard HyperTransport

HyperTransport is used by AMD Opteron [35] to interconnect the processors in a motherboard. In these systems, each processor requires to know where a given memory request must be forwarded. This is achieved by including in each processor a set of registers configured at the initialization phase that reflect the system physical memory distribution. In this way, when a processor issues a memory operation (load or store) on a given memory location, the processor compares the requested address with the contents of the mentioned registers and then, depending on the results, a memory request is forwarded to the memory controller handling the memory address. The action of forwarding the memory operation involves the generation of a HyperTransport message.

3.1.3 Connecting Motherboards to Access Remote Memory

A process must be able to access not only the memory installed on its board but also the memory installed in other boards. To this end, additional hardware support is required to provide fast access. The so-called RMA (Remote Memory Access) hardware implements the required functionality. This hardware component is seen by the processors in the motherboard as a new memory controller. However, the RMA does not act as a typical memory controller since it has no memory banks directly connected to it, otherwise it relies on the memory banks installed in other nodes of the cluster. To enable the functionality, the registers mentioned above must be reconfigured so that some of the memory accesses (i.e., those accessing the memory installed in other motherboard) are directly forwarded to the RMA, which converts those accesses into remote accesses. The RMA has a regular HyperTransport interface to the local node and a High Node Count HyperTransport [36] interface to the rest of the cluster and it is attached to the motherboard in the cluster by means of HTX compatible cards [8].

3.2 System Model

3.2.1 System Model Characteristics

To focus the research, we concentrate the analysis and scheduling policies in two boards, that is, we use a scaled system consisting of two motherboards, each one composed of a 2-node NUMA system as shown in Figure 3.2. As can be seen, each node includes a processor with private caches (i.e., L1 and L2), its memory controller and the associated RAM memory.

3.2.2 Memory Regions

Three different regions of memory in the system have been considered:

- *Local to Node:* Memory located in the processor in which the application is launched.
- *Local to Board:* Memory located on the same board but attached to the other processor.
- *Remote:* Memory located in the other motherboard.

Figure 3.2 shows the modeled system. The cluster computer consists of two boards, namely local and remote, connected by means of a high speed interconnection network and the RMA to keep a low access time to remote memory.

The local board models two nodes which are composed of a single-issue processor and local memory, while DRAM memory is the only resource used from the remote board and has not been represented. Remote memory is assumed to be partitioned into two parts: i) a portion which is used by the processors in the motherboard, and ii) the exceeding memory is considered to be available for the OS installed in other boards. In other words, the local OS is able to see both the local and the remote memory. The machine parameters concerning both the processor and the memory that have been considered for experimental purposes are summarized in Table 3.1.

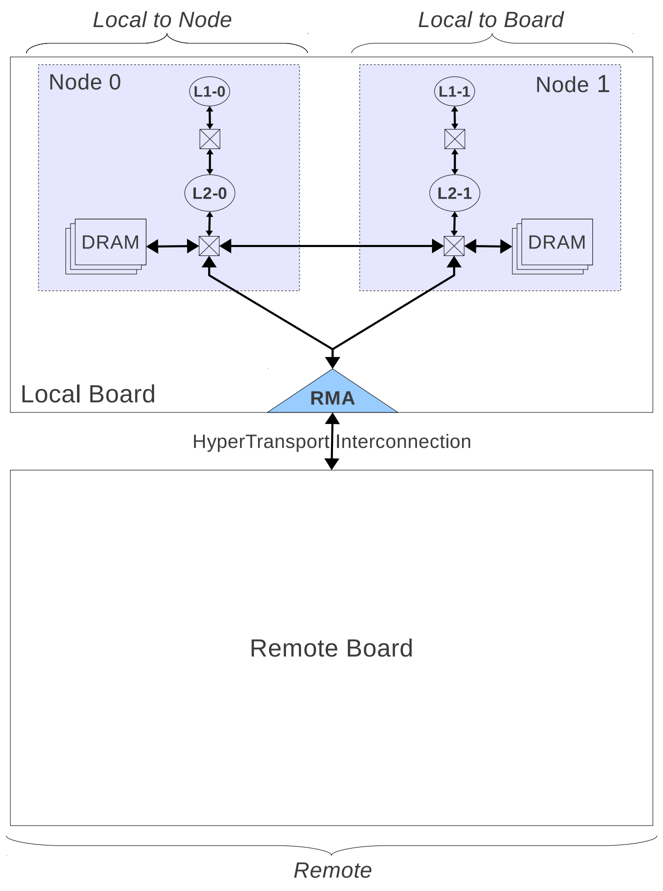


FIGURE 3.2: System model.

3.2.3 Memory Scheduling

As the memory regions in the system (i.e., Local to Node, Local to Board, Remote) present different latencies, performance of a given application strongly depends on how its assigned memory is distributed among these regions. Since each application contributes with its performance to the global performance, a memory scheduler is required to maximize the global performance. This memory scheduler must be aware not only of the characteristics (i.e., latency, bandwidth) of the different memory regions

Table 3.1: Machine Parameters.

Component	**Parameter**
Issue width	single issue
Issue policy	out-of-order
L1 cache: size, #ways, line size	64KB, 2, 64B
L1 cache latency	1 cc
L2 cache: size, #ways, line size	1MB, 16, 64B
L2 cache latency	6 cc
Local memory latency	100 cc
Remote memory latency	410 cc

but also of the memory requirements of the running applications. For example, allocating by 25% of the available remote memory to a memory-intensive application could yield to worse performance than allocating the whole remote memory to an application showing good cache locality.

To choose a good distribution of the different memory regions among the running applications, the scheduler requires information about the expected performance for a given memory distribution. This dissertation focuses on the design of efficient memory-aware scheduling policies.

3.3 Workloads

The benchmark suites executed in the experimental evaluations are SPEC CPU2006 [37, 38] and SPLASH2 [39]. Additionally, we have executed the Stream benchmark [40], since its memory features have been considered of interest for the purpose of this research. A brief description of these programs is given next, citing the command-line arguments for each benchmark.

3.3.1 Stream

The STREAM benchmark is a synthetic benchmark program, written in standard Fortran 77 (with a corresponding version in C). It measures the performances of four long vector operations: (i) *Copy* measures transfer rates in the absence of arithmetic;

(ii)*Scale* adds a simple arithmetic operation; (iii)*Sum* adds a third operand to allow multiple load/store ports on vector machines to be tested; and (iv)*Triad* allows chained/overlapped/fused multiply/add operations.

These operations are representative of the "building blocks" of long vector operations. The array sizes are defined so that each array is larger than the cache size of the machine to be tested, and the code is structured so that data re-use is not performed. The intent of STREAM is not to suggest that "real" applications have no data re-use, but rather to decouple the measurement of the memory subsystem from the hypothetical "peak" performance of the machine.

3.3.1.1 Determining the Stream Array Size

Before using Stream in our experiments, a study to select the most suitable problem size for the simulations was performed. To this end, a simple system with only one motherboard containing two processing nodes (node 0 and node 1) and the main memory allocated in one of these two nodes was defined. Stream benchmark was executed in such a system varying two parameters:

- the *local node* (where the application runs) and the *remote node* (the other node in the system)
- the node which contains the main memory

The two configuration alternatives in such a system are listed below:

i) Application is running in node 0 and the main memory is located in node 0.

ii) Application is running in node 0 and the main memory is located in node 1.

Note that when the local node is not the node that contains the main memory of the system, the application needs to access memory out of its local node when the required data is not in cache memory. Consequently, this situation yields to worse performance than executing the application in the node which contains the main memory of the system. An experiment associated to each of the two aforementioned cases was executed. In this scenario, the performance difference between cases *i* and *ii* was calculated. This mechanism was carried out for different array sizes, looking for the smallest size, $size_x$, which fulfilled the following requirements:

- Achieving the highest performance difference between cases *i* and *ii*.
- Leading to a performance difference between $size_x$ and the immediately following size, $size_{x+1}$, as similar as possible (i.e., stabilization) for consecutive sizes.

First, a coarse grain study was performed to reduce the search range for the array size. Once the Stream array size was delimited within a range from 55000 (i.e., $size_1$) to 495000 (i.e., $size_9$) elements, we carried out a fine grain analysis among nine array sizes (with a difference of 55000 between each size and the following) within this interval whose results are shown in Table 3.2. From these results, concluded that 275000 is the most suitable problem size (i.e., $size_5$) for our simulations, since it is the smallest array size which leads to performance difference stabilization between two consecutive problem sizes (the performance difference achieved by sizes 275000 and 330000 is the most similar). The smallest suitable size has been chosen as simulation time considerably grows with the array size.

3.3.2 SPLASH-2

SPLASH-2 suite is composed of a set of 11 shared-memory, multi-threaded benchmarks, which are classified as kernels or applications. SPLASH2 benchmarks perform computations, synchronizations, and communication, stressing processor cores, memory hierarchy, and interconnection networks. They constitute a widely used evaluation tool for the research community.

For simulation purposes, a set of three SPLASH2 benchmarks have been chosen for our experiments. Our work focuses in memory access, thus the benchmark parameters which are considered in order to select the set of executed workloads are:

Table 3.2: Stream size fine grain analysis.

x	log_2 **array size**	**local node**	**memory node**	**performance difference (%)**
1	55000	0	0 1	26.73
2	110000	0	0 1	25.89
3	165000	0	0 1	26.08
4	220000	0	0 1	26.23
5	**275000**	0	0 1	**26.32**
6	330000	0	0 1	26.38
7	385000	0	0 1	26.42
8	440000	0	0 1	26.46
9	495000	0	0 1	26.48

- Number of executed instructions ($\#instr$).
- Number of executed memory read ($\#loads$) instructions.
- Number of executed memory write ($\#stores$) instructions.
- Cache Miss Rate ($Cache_{miss\,rate}$).

All the benchmarks of the suite have been evaluated in terms of the memory criterion defined in equation 3.1, which aims to quantify the amount of *memory stress* performed by a given benchmark during its execution.

$$memory\ stress = \frac{\#loads + \#stores}{\#instr} \times Cache_{miss\,rate} \tag{3.1}$$

The quotient $\frac{\#loads + \#stores}{\#instr}$ is the percentage of memory instructions executed. Since most accesses hit the cache, we multiply this term by the cache miss rate in order to obtain the fraction of memory reference instructions issued by the processor that effectively access to main memory.

For each benchmark, the *memory stress* has been calculated (see Table 3.3). Those benchmarks performing a higher memory stress are the most suitable for our study.

From the results of this evaluation, we conclude that the best SPLASH2 workloads for our goal are Radix ($memory\,stress = 0.636$), FFT ($memory\,stress = 0.240$), and Cholesky ($memory\,stress = 0.1982$), since they achieve the highest values for the established criterion.

TABLE 3.3: Splash2 benchmarks characteristics and associated memory stress.

Benchmark	**#instr**	**#loads**	**#writes**	**mem. instr.** (%)	$cache_{miss\,rate}$	**memory stress**
Barnes	2002,79	406,85	499,72	0,36	0,05	0.0180
Cholesky	539,17	313,29	284,61	0,26	36,0	0.182
FFT	34,79	111,86	12,06	0,2	0,7	0.24
FMM	1250,02	28,03	7,03	0,21	97,9	0.051
LU	494,05	4,07	208,9	0,31	1,2	0.034
Ocean	379,93	2,88	79,95	0,27	8,3	0.167
Radiosity	2832,47	226,23	152,19	0,28	0,24	0.047
Radix	50,99	38,58	59,57	0,37	63,6	0.636
Raytrace	829,32	104,00	81,27	0,35	0,11	0.122
Volrend	754,77	48,00	35,25	0,28	16,7	0.101
Water-Nsq	460,52	81,89	72,31	0,25	0,63	0.025
Water-Sp	435,42	18,93	32,73	0,24	63,5	0.022

3.3.2.1 Describing the Selected SPLASH2 Benchmarks and their Problem Sizes

All the SPLASH2 benchmarks provide command-line arguments or configuration files to specify the input size. Since the problem size can be flexibly tuned to provide reasonable simulation times, programs are run until completion. In this section, each selected SPLASH2 benchmark is briefly described and the used command-line arguments are specified for each case.

FFT

The FFT kernel is a complex, one-dimensional version of the Six-Step FFT [41]. Several command-line parameters must be specified: the number of points to transform, the number of processors, the log base 2 of the cache line size, and the number of cache lines. Both the number of data points and the number of processors must be a power of 2.

The method for choosing the most suitable matrix size (i.e., number of points to transform) for FFT benchmark has been the same as the one explained in Section 3.3.1.1 for Stream benchmark, whose detailed results were shown for illustrative purposes. The same methodology has been used to obtain the problem sizes of the remaining SPLASH2 benchmarks (i.e., Radix and Cholesky)

From the problem size study of FFT, we selected $2^{16} = 65536$ total complex data points transformed.

Regarding the rest of the command line options, we have used the following:

- 1 processor (default value)
- 1024 cache lines.
- $2^6 = 64$ cache line length in bytes.

Radix

The RADIX kernel implements an integer radix sort based on the method described in [42]. Several command line parameters can be specified. The number of keys to sort, the radix for sorting, and the number of processors are those parameters that are normally changed. The radix used for sorting must be a power of 2. The values of these parameters are listed below:

- 1 processor (default value)
- Radix for sorting (Must be power of 2)= 1024 (default value)
- Number of keys to sort (Must be power of 2)= 524288= 1M/2
  - The value of this parameter used for evaluation in [39] is 1M integers. However, when executing the application with this size, we got excessively large simulation times, while minimal performance differences appear in comparison studies with respect to 500K integers, which has been the used value in this work.

- Maximum key value= 524288(default value). Integer keys k are generated such that $0 <= k <= 524288$.

Cholesky

The blocked sparse Cholesky factorization kernel factors a sparse matrix into the product of a lower triangular matrix and its transpose.

The size of the cache (in bytes) should be specified on the command line, as well as the number of processors being used. The postpass partition size should be kept at the default value of 32.

- Postpass partition size= 32(default value)
- Cache size in bytes= 65536
- 1 processor (default value)
- Input file= tk15.O

3.3.3 SPEC CPU2006

SPEC CPU2006 focuses on performance of the processor (CPU), the memory architecture, and the compilers. This suite has some main advantages over other benchmarks suites: i) it is developed from actual end-user applications (the benchmarks are not synthetic), ii) it is widely supported by vendors, and iii) it is highly portable.

The suite is formed by 29 single-threaded benchmarks, classified as integer or floating-point. Integer benchmarks (see Table 3.4) are written in C or in C++, and include compression, compilation, artificial intelligence algorithms, XML Processing, or path-finding algorithms, among others. Floating-point benchmarks (see Table 3.5) are written in C, C++, Fortran or a mixture of C and Fortran, and deal with physics simulation, image processing, weather prediction, speech recognition, etc. For each benchmark, three input data sets with different sizes are provided, named *test*, *train*, and *ref*.

Table 3.4: Command-line arguments for the SPEC2006 Integer benchmarks.

Integer Benchmark	**Arguments**
400.perlbench	*checkspam.pl* 2500 5 25 11 150 1 1 1 1
401.bzip2	*input.source* 280
403.gcc	166.*i* -o 166.*s*
429.mcf	*inp.in*
445.gobmk	--*quiet* --*mode gtp* < *trevord.tst*
456.hmmer	--*fixed* 0 --*mean* 500 --*num* 500000 --*sd* 350 --*seed* 0 *retro.hmm*
458.sjeng	*ref.txt*
462.libquantum	1397 8
464.h264ref	-*d sss_encoder_main.cfg*
471.omnetpp	*omnetpp.ini*
473.astar	*rivers.cfg*
483.xalancbmk	-*v t*5.*xml xalanc.xsl*

Table 3.5: Command-line arguments for the SPEC2006 Floating-point benchmarks.

Floating-point	**Arguments**
410.bwaves	–
416.gamess	< *triazolium.config*
433.milc	< *su*3*imp.in*
434.zeusmp	–
435.gromacs	-*silent* -*deffnm gromacs* -*nice* 0
436.cactusADM	*benchADM.par*
437.leslie3d	< *leslie*3*d.in*
444.namd	--*input namd.input* --*iterations* 38 --*output namd.out*
447.dealII	23
450.soplex	-*m*3500 *ref.mps*
453.povray	*SPEC*-*benchmark*-*ref.ini*
454.calculix	-*i hyperviscoplastic*
459.GemsFDTD	–
465.tonto	–
470.lbm	3000 *reference.dat* 0 0 100_100_130_*ldc.of*
481.wrf	–
482.sphinx3	*ctlfile* . *args.an*4

All the benchmarks in the SPEC2006 suite are used in this thesis to evaluate several memory distributions under different memory scheduling policies. For simulation purposes, a set of the first 750M committed uops has been considered a representative execution of the benchmarks, thus we have run detailed simulations for each benchmark until this point and the computed the final statistics. The files for the input data are taken from the *ref* set. For each benchmark, Table 3.4 and Table 3.5 list the command-line arguments used for its execution.

3.4 Simulation Tool

The selected simulator must be able to model current processors in order to provide representative results. Current microprocessors are really complex, thus the simulator should provide a relatively easy to understand interface as well as facilitating the extraction of statistics results. The fact of including some development support such as debugging or graphic trace tools is also useful for implementation and checking the validity of the implementation. Finally, the simulator must be able to execute current industry-standardized, intensive workloads and the benchmark suites most widely used by the research community. Among the available simulators, we have chosen Multi2Sim simulation framework, which was originally developed in our research group and is used by many researchers in companies and universities around the world. Multi2Sim is a detailed cycle-by-cycle execution driven simulator for superscalar, multithreaded, multicore, and graphics processors [43].

3.4.1 Multi2Sim

Multi2Sim project is as a free and open-source tool which shares its settled and ongoing research activities with the community. With the main simulator and its satellite tools published on its website [44], the project intends to create a unified, automated, and publicly accessible methodology for computer architects to evaluate their alternative design proposals. The integration eliminates the need to individually carry out frequent and routine activities, such as setting up tedious simulation infrastructures or architectural exploration scripts. The open-source nature of Multi2Sim and its complete documentation brings the architecture of state-of-the-art processors closer to new researchers, and allows instructors to use it as a pedagogical tool to teach, illustrate, and evaluate a wide range of CPU/GPU-related concepts. Multi2Sim is an application-only simulator, which allows one or more applications to be run on top of it without booting a guest operating system first. Its main characteristics are listed below.

3.4.1.1 Simulation Features

Superscalar Pipelines

Multi2Sim supports a cycle-based simulation of superscalar pipelines, modeling instruction fetch, decode, issue, write-back, and commit stages. The model features detailed microarchitectural structures such as the reorder buffer, load/store queues, register file, or trace cache. The pipeline front-end supports different types of branch prediction and micro-instruction decoding, while the back-end implements out-of-order and speculative execution.

Multithreading

A multithreaded pipeline model supports execution of multiple programs or one parallel application spawning child threads. The multithreaded processor shares a common pool of functional units among hardware threads. The rest of the pipeline resources can be configured as private or shared among hardware threads. Multi2Sim supports models for coarse-grain, fine-grain, and simultaneous multithreading.

Multicore

Superscalar and multithreaded pipelines are replicated a configurable number of times forming models of multicore processors. Cores communicate through the memory hierarchy with transactions triggered by the memory coherence protocol.

Graphics Processing Units

A cycle-based simulation model is provided for state-of-the-art AMD and NVIDIA Graphics Processing Units (GPUs). Multi2Sim can run unmodified OpenCL programs, intercepting OpenCL function calls, transferring control to a custom runtime, and launching simulation of OpenCL device kernels. Original host and kernel binaries can run on Multi2Sim without an actual GPU being installed on the system.

Memory Hierarchy

The memory hierarchy is modeled with an event-driven simulation of cache memories, organized with a configurable number of cache levels, geometries, and latencies. An implementation of the MOESI protocol handles coherence between caches from different cores. The model also features directories for caches and main memory. The upper level caches can belong to different threads, different cores, or a subset of them, and they can be both instructions and data caches. Even if one instruction cache and one data cache are connected to a unified cache at the lower level, coherence is maintained among them (which is required in the case of, for example, self-modifying code).

Interconnection Networks

Components in the memory hierarchy communicate through interconnection networks, with configurable topologies, link bandwidths, routing algorithms, and virtual channels. An automatic cycle detection mechanism warns about possible deadlock condition in networks.

Heterogeneous Computing

Multi2Sim integrates models for different CPU and GPU architectures, all of them simulated at the ISA level for high accuracy purposes. This integration allows researchers to evaluate configurations of state-of-the-art commercial processors, where heterogeneous processing devices are encapsulated in the same die.

Context Scheduling

Multi2Sim introduces the concept of context scheduling after version 2.3.3, similar to the idea of process scheduling in an operating system. The scheduler is aimed at mapping software contexts to processing nodes (hardware threads) to run them. There are two types of context scheduling: static and dynamic.

Simulation Paradigm

The simulation paradigm can be divided into two main modules: the functional simulation and the timing or detailed simulation. The functional simulation is just an emulation of the input program. Given an executable ELF (*Executable and Linkable Format*) file, the functional simulator provides the same behavior as if the program was executed natively on an x86 machine. The detailed simulator provides a model of the hardware structures of an x86-based machine; it provides timing and usage statistics for each hardware component, depending on the instruction flow supplied by the functional simulator.

Statistics Reports

Multi2Sim offers the possibility of generating several statistics reports, global or specific, with all kind of variables which can be used at researching tasks.

The CPU Statistics Summary of a program simulation on Multi2Sim reports information such as Total simulation time in seconds, Total number of simulated instructions, Quotient of Instructions and Time, Maximum number of active contexts during the simulation, Maximum amount of memory in bytes used in total by all contexts during the simulation, Number of CPU cycles simulated, Quotient of Instructions and Cycles, Percentage of branches correctly predicted, or Quotient of Cycles and Time.

On the other hand, a detailed report about the stages of the processor pipeline can be obtained. The statistics are classified by thread and execution node. There are global results for all the stages of the pipeline, and specific results for either hardware structures (reorder buffer, instructions queues, branch target buffer) or each single pipeline stage.

The memory hierarchy statistics contains one section per cache, main memory module, and interconnect. For each interconnect, the statistics report includes those sections and variables specified in the description for the network. The information related to the caches include accesses, misses, hits, reads, writes, etc. Likewise, the interconnection network simulation report is composed of transfers, messages, bandwidth, utilization, among others.

As the simulation framework grows, the configuration files and results reports format are improved, as well as more detailed, usable and graphic new debugging tools are added.

The simulation environment provides user-friendly and easy to use debugging tools. For instance, the pipeline debugger is used to view graphical timing diagrams for detailed simulations, in which the pipeline state can be traced cycle by cycle with a representation of executed macroinstructions and uops. This tool is based on the *ncurses* library, which allows graphical representations on terminals by using plain text characters.

In addition, the Multi2Sim GPU model has its own pipeline debugger. The GPU pipeline debugger is a tool that provides analysis of OpenCL code run on Multi2Sim's GPU architectural simulator. Using a graphical user interface, the pipeline debugger acts as a visual aid in testing code and conceptualizing the architecture of the simulator itself. Its primary function is to observe the state of the GPU pipeline, providing features such as simulation pausing, stepping through cycles, and viewing properties of in-flight instructions.

Multi2Sim uses INI files format for all of its input and output files, such as detailed simulation statistics reports, context configuration or cache hierarchy configuration files.

Finally, Multi2Sim has been adapted to provide those statistics that McPAT requires in its input file. McPAT is an integrated power, area, and timing modeling framework that supports comprehensive design space exploration for multicore and manycore processor configurations ranging from 90nm to 22nm and beyond. Though the processor models provided by McPAT and Multi2Sim are not exactly the same, still some common configurations can be obtained to estimate the global energy dissipated for a given benchmark execution.

3.4.2 Simulation Tool Extensions

As mentioned above, Multi2Sim is an open source simulation framework. All the scheduling policies proposed in this work as well as the underlying processor and system architectures on which they have been evaluated, have been modeled on an extensively modified version of the Multi2Sim simulator. Among the main developed extensions are:

In Order Execution

The Issue stage of the processor pipeline has been extended with the possibility of choosing an in-order execution paradigm, since Multi2Sim natively only simulates out-of-order execution processors. If an in-order execution is chosen, we control the different queues, that is, instruction queue (IQ) and load and store queue (LSQ), so that the oldest instruction (i.e., the one which owns the lowest sequence number) is issued first.

Memory Network Hierarchy

A memory network hierarchy which fits our local and remote memory access model has been devised. The available interconnection elements of Multi2Sim have been adapted to simulate our memory system. A network node has been placed for each cache memory and each main memory module. Then different network configurations have been used to simulate the connections between them, taking into account the message sizes and the goal latencies for each cache level and the three considered memory regions (i.e., L, Lb, and R).

Memory Management Unit

The memory management unit has been completed with some additional functionalities related to different memory placement and replacement policies in order to analyze how to keep the most accessed pages in the local memory region and thus improving the system performance. In this context, we have devised our own Translation Lookaside Buffer (TLB) and implemented several page-granularity memory distribution schemes as well as some main memory scheduling algorithms.

Memory Size

As Multi2Sim assumes a non-limited main memory space, a mechanism to constrain that size and determine the available memory size has been included for each execution and single application.

Quality of Service

On the other hand, Quality of Service control methods have been implemented to avoid situations in which the performance of an application is damaged under a given threshold value.

3.5 Summary

This chapter has introduced the system model and experimental framework of the thesis. First, the target system has been presented. The baseline cluster machine and the interconnection network technology have been detailed to finally describe the mechanism by means of which the main memory of a remote motherboard of the cluster is accessed. Then, the system model has been explained, in particular the memory subsystem and the different main memory regions. Finally, the experimental framework has been defined. On one hand, the different workloads executed in the experimental evaluations have been briefly presented, including the command-line arguments for their execution. On the other hand, the Multi2Sim simulation framework and our extensions to this tool have been described.

Chapter 4

Scheduling Policy based on a Cost-effective Heuristic

The execution time of a huge amount of current applications mainly depends on how efficiently the system handles memory accesses. In the system described in Section 3.1, memory management is a critical performance issue since the memory used by the applications can reside in three main locations (i.e., L, Lb, R) that have different latencies. Notice that, in this way, obtaining additional DRAM memory suffers from higher latencies for remote memory. Thus, an efficient memory scheduling is required to obtain the best performance in this scenario.

In this context, this chapter proposes two memory scheduling algorithms (an ideal algorithm and a heuristic strategy) that assign memory from the three memory locations mentioned above to applications in order to provide the best overall system performance, while guaranteeing a minimum quality of service (QoS) to each application. To this end, we first explore how the application performance behaves depending on its memory requirements in order to identify critical aspects that could help the design of memory scheduling algorithms. In other words, we analyze, through different workloads, how the distribution of memory accesses among the three different memory locations impacts on the applications performance.

We assume that the distribution of local and remote memory assigned to an application is set statically by interleaving the memory addresses at cache block size level (64B). Consequently, the memory controllers are configured when the system boots to support a given distribution.

4.1 Analysis and Impact on Performance of Memory Distribution

This section analyzes the impact on performance when varying the memory distribution across the three memory regions (L, Lb, R). Four benchmarks have been used to carry out the experiments: Stream [40] and three kernels (Radix, FFT and Cholesky) from the SPLASH-2 benchmark suite [39]. Stream is a benchmark designed to stress the memory hierarchy, while the selected SPLASH-2 kernels have been chosen because they perform the highest number of memory accesses of the benchmark suite.

First, we study the case that only the DRAM installed in the local board (i.e., L and Lb) is allocated to an application. The performance (i.e., IPC) has been analyzed varying the percentage of Lb with respect to the total assignment (i.e., $L + Lb$). Figure 4.1(a) shows the results. Since only L and Lb modules are assigned, it is enough to represent the Lb percentage in the X axis. For instance, a value of 75 in the X axis corresponds to an assignment of $Lb = 75\%$ and $L = 25\%$. Notice that the distribution of local memory assignment may have a strong impact on the performance of some benchmarks while others are slightly affected. Stream is the most sensitive benchmark since its IPC degrades about 42% when all its memory is assigned to the Lb memory region, while FFT performance degrades about 27%, and Radix and Cholesky performance hardly degrades.

Table 4.1: Impact on IPC for different memory distributions.

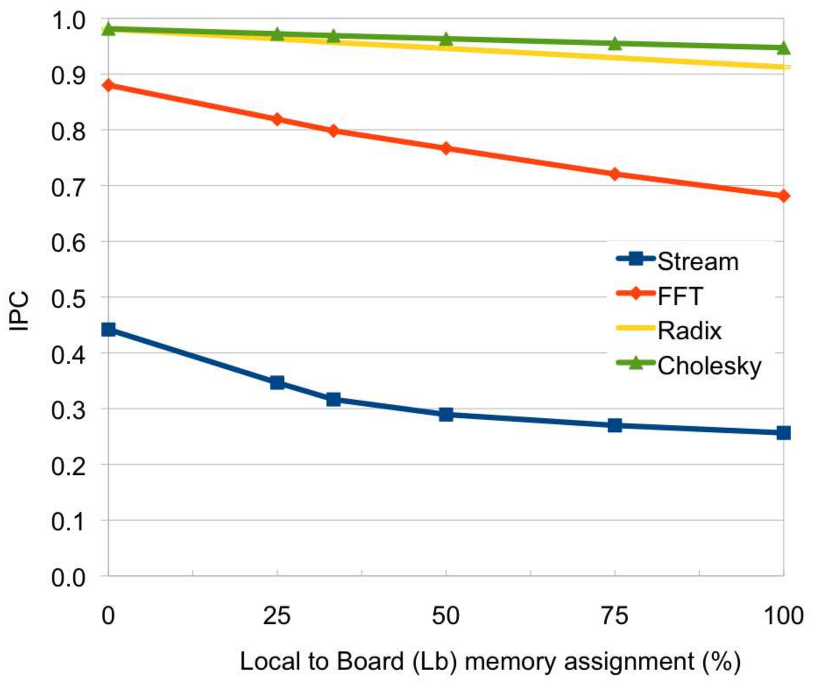
Memory Distribution				IPC			
L(%)	**Lb(%)**	**(L+Lb)(%)**	**R(%)**	**Stream**	**FFT**	**Radix**	**Cholesky**
50	25	75	25	0,06	0,42	0,73	0,85
25	50			0,06	0,40	0,72	0,85
33,3	33,3	66,7	33,3	0,06	0,39	0,71	0,84
66,7	0			0,06	0,39	0,70	0,84
50	0	50	50	0,05	0,29	0,60	0,78
33,3	16,7			0,05	0,29	0,59	0,77
25	0	25	75	0,04	0,22	0,50	0,71
0	25			0,04	0,22	0,50	0,71

In the second scenario, the impact of allocating remote memory is explored. Table 4.1 shows the performance results for eight different memory distributions. Notice that the performance slightly differs when varying the memory distribution of L and Lb while maintaining their accumulated value (dark cell) constant. This means that the execution time is dominated by the much slower remote memory. For instance, for $L + Lb = 75\%$, Stream gets the same IPC (0.06) in both cases. This is because remote memory has a latency about one order of magnitude higher than local memory. From these results, we can conclude that, when assigning remote modules, the memory distribution within the board (L and Lb) has a negligible impact on performance. Consequently, to study the effect of remote memory, a single value is enough to represent both local memory regions (i.e., $L + Lb$ column).

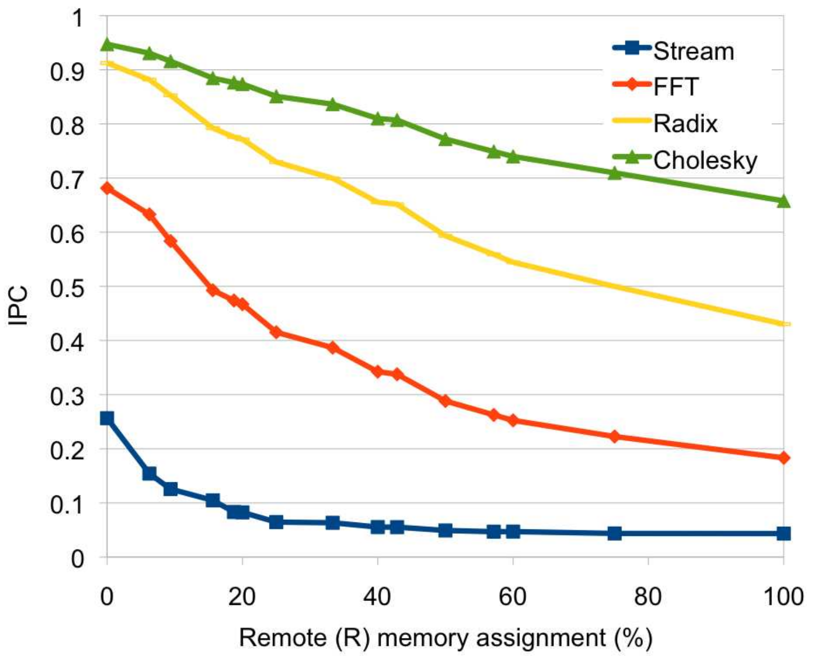
Figure 4.1(b) shows the adverse impact on performance as the percentage of assigned remote memory grows with respect to the total memory assignment (i.e., $L+Lb+R$). The initial points in the Y axis correspond to the lowest performance of each application in Fig. 4.1(a). Values in the X axis represent the percentage of remote memory assignment. For instance, $X = 25$ means that $R = 25\%$ and $L + Lb = 75\%$. Three different performance behaviors can be appreciated. Stream performance dramatically degrades with the assigned remote memory until around 25%, where it stabilizes. On the other hand, performance of both Radix and Cholesky constantly decreases in almost a linear way. Finally, FFT behavior falls in between these two trends. Its performance strongly drops until $R = 60\%$ and then it smoothly decreases.

4.2 Concurrent Execution of Several Applications

When running several applications, it may happen that the amount of remote memory allocated to a given application yields to unacceptable performance for that application. This section first presents a Quality of Service (QoS) parameter to deal with such situations. This parameter specifies the maximum acceptable performance degradation for an application.



(a) Impact on IPC when assigning L and Lb



(b) Impact on IPC when assigning R

FIGURE 4.1: Impact on performance (IPC) when assigning local and remote memory.

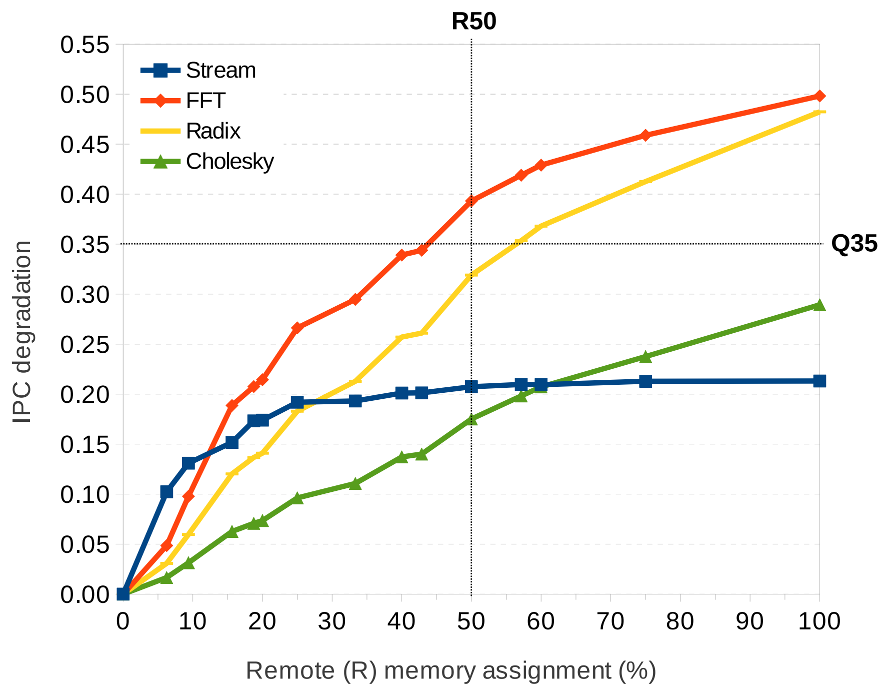


FIGURE 4.2: IPC degradation when varying the percentage of assigned remote memory.

4.2.1 Quality of Service Definition

Figure 4.2 shows the IPC degradation caused by assigning a given $X\%$ of remote memory and calculated as $IPC(R = 0\%) - IPC(R = X\%)$. The origin point means that no remote memory is assigned so there is no performance degradation. Notice that since performance always degrades as R increases, this figure can be used to define the maximum percentage of remote memory that can be allocated to an application in order to to guarantee an acceptable (QoS) performance. For instance, if the maximum IPC degradation permitted for FFT is 0.35, the scheduler will not assign more than 40% of remote memory to this benchmark (see label Q35). On the other hand, if the percentage of remote memory allocated to Cholesky is less than 50%, then its IPC degradation will be below 0.18 (see label R50). Therefore, there is a bidirectional relationship between performance degradation and percentage of assigned remote memory. Due to this equivalence, from now on, the maximum percentage of remote memory that is permitted to be assigned to a given application will be referred to as its QoS, and

Table 4.2: Studied memory distributions.

Application	Case A		Case B		Case C		Case D		Case E	
	(L+Lb)	R	(L+Lb)	R	(L+Lb)	R	(L+Lb)	R	(L+Lb)	R
A1	100%	0%	75%	25%	50%	50%	25%	75%	0%	100%
A2	0%	100%	25%	75%	50%	50%	75%	25%	100%	0%

this value will be used by the devised memory schedulers for controlling the system performance.

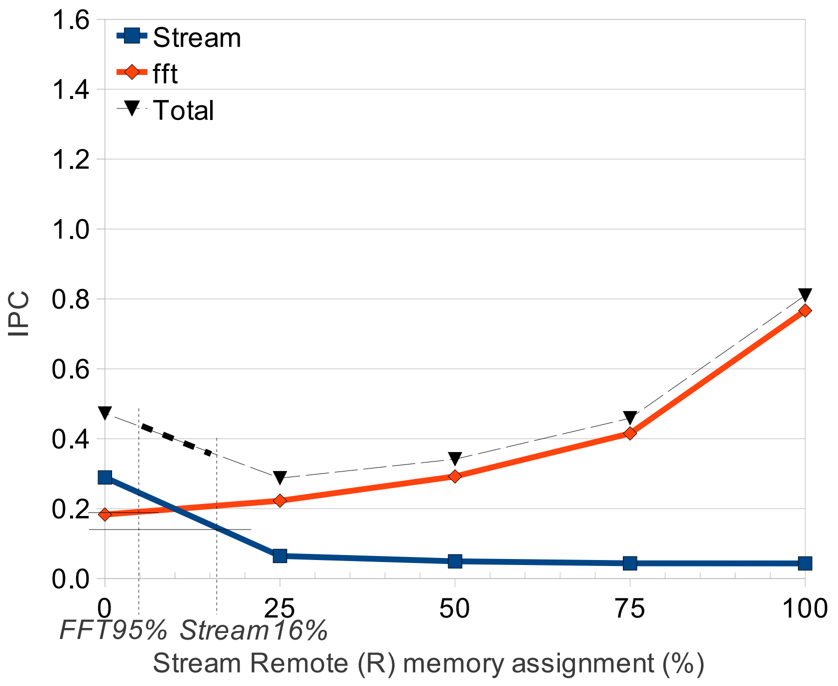
4.2.2 Two Concurrent Applications

Once the QoS has been defined, we analyze the impact of the memory distribution on the performance of several concurrent applications. In this section, the study focuses on two applications. Two cases are analyzed: in the first one, the whole remote memory is allocated, and in the second one, only a fraction of it.

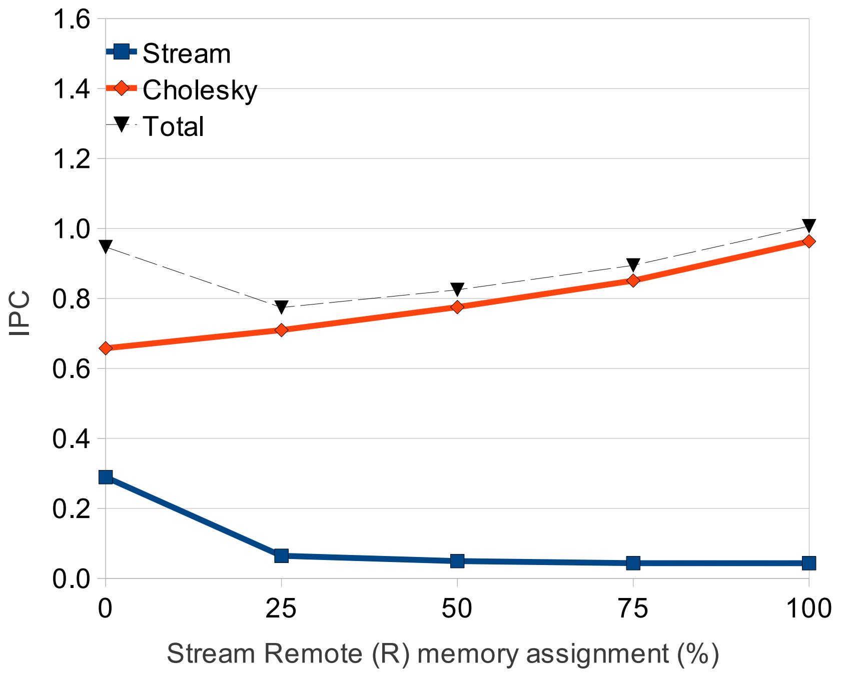
4.2.2.1 Complete Usage of Remote Memory

This study assumes that all the memory installed in two motherboards (local and remote) is used by two applications and that there is enough memory to support their whole working set. Five memory distributions have been evaluated varying the percentage of remote memory assigned to each application from 0% to 100% in fractions of 25%, as shown in Table 4.2. For instance, case B means that the memory assignment for application A1 is R=25% and L+Lb=75% while the application A2 allocates R=75% and L+Lb=25%. Using these distributions, the performance of each application as well as the combined performance are shown in Figure 4.3 and Figure 4.4.

Since remote memory is shared between both applications, if one of them uses $R = X\%$, then the other one shall use $R = 100\% - X\%$. For instance, in Figure 4.3 (a) if Stream consumes $R = 25\%$ then FFT will use the remaining $R = 75\%$. The dashed line stands for the total system performance. The highest point represents

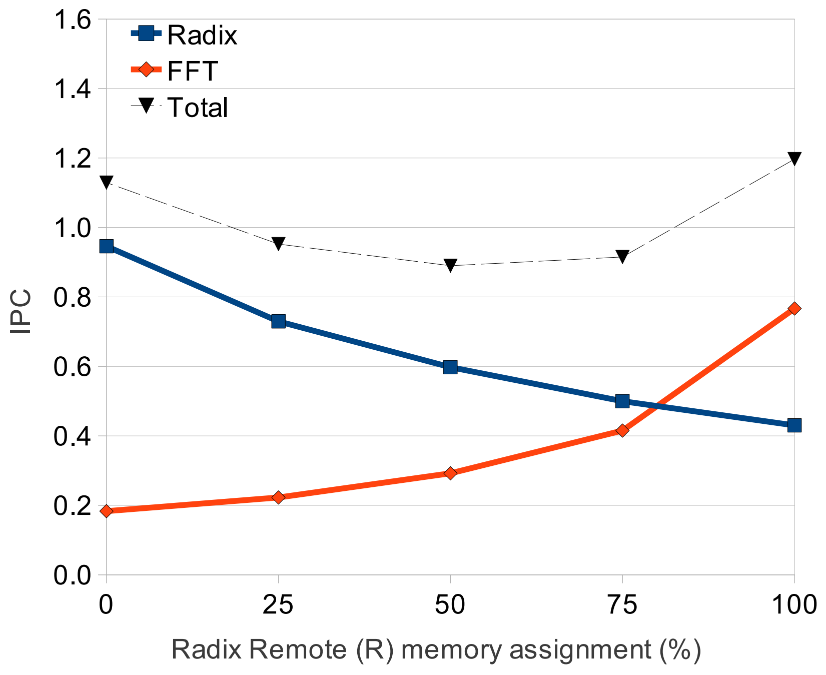


(a) Stream vs FFT

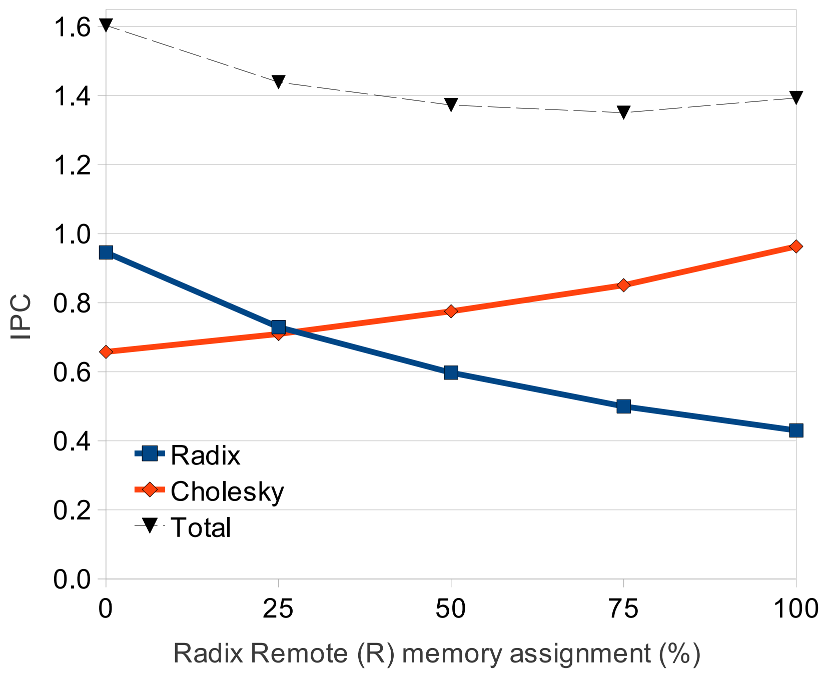


(b) Stream vs Cholesky

FIGURE 4.3: Overall IPC for couples of applications.
Stream versus FFT and Cholesky.



(a) Radix vs FFT



(b) Radix vs Cholesky

Figure 4.4: Overall IPC for couples of applications.
Radix versus FFT and Cholesky.

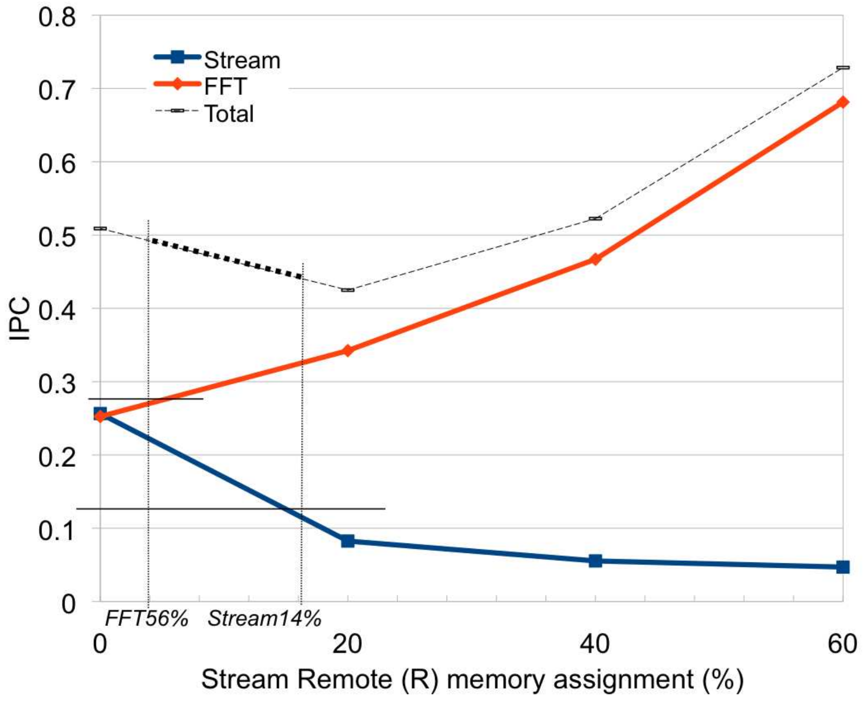
the maximum performance achieved by both workloads; however, reaching this point might imply poor and unacceptable performance for some applications. This situation can be controlled by defining a QoS for each application as stated above. For instance, if Stream QoS is set to 16% and FFT QoS is set to 95% (see labels *Stream16%* and *FFT95%* in Figure 4.3 (a)), the scheduler will select the distribution corresponding to the aggregated $IPC = 0.45$ (i.e., Stream consumes $R = 5\%$ and FFT uses the remaining $R = 95\%$), since it is the best performance falling in the interval [0.35, 0.45] defined by the QoS restrictions of both benchmarks. Consequently, the IPC of Stream will be five times higher than its worst case (0.25 versus 0.05).

Notice that as the distribution of R changes, the performance of one application increases while the performance of the other drops. Since this fact happens at different rates, the maximum performance is reached when the application that decreases performance at the highest rate (i.e., the most sensitive one) only accesses local memory. This application can be easily identified by looking its IPC degradation value for $R = 100$ in Figure 4.2. For example, if the two co-running applications are Stream and FFT, their IPC values will be 0.21 and 0.50, respectively. Thus, the maximum aggregated IPC is obtained when all the remote memory is assigned to Stream.

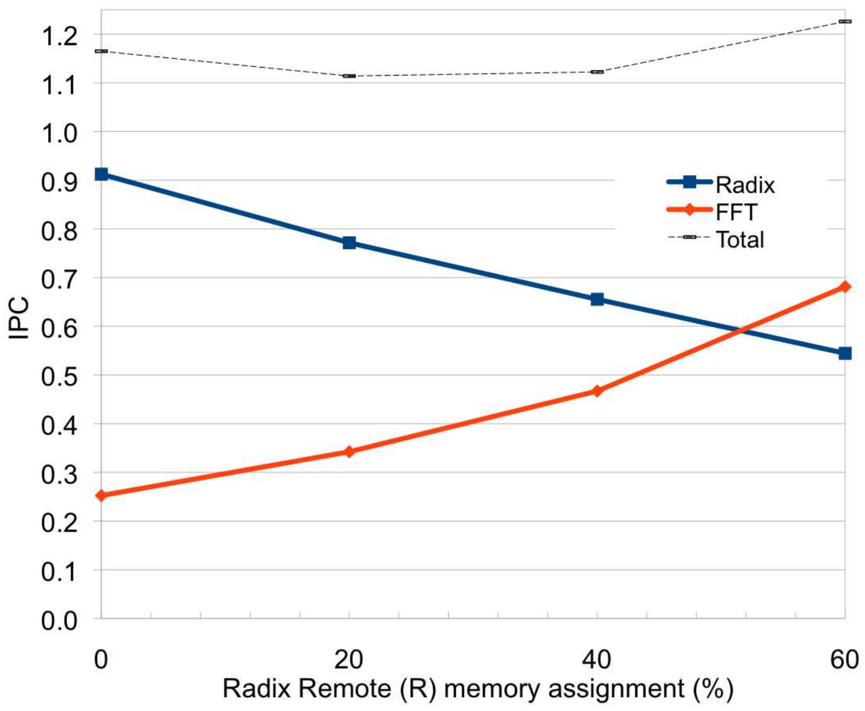
4.2.2.2 Partial Usage of Remote Memory

This section analyzes the case where the applications use all the local memory but only a fraction of the available memory in the remote motherboard. For illustrative purposes, we consider the scenario where 60% of remote memory is used by the local applications. That is, if an application uses $X\%$ of remote memory, the other one will use $60\% - X\%$. Notice that since the amount of remote memory consumed is less than 100%, the overall performance should be better than in the previous study.

Figure 4.5 shows the results for two couples of benchmarks: Stream and FFT (Figure 4.5 (a)), and Radix and FFT (Figure 4.5 (b)). To plot the IPC, both applications have been profiled for different percentages of remote memory, the extreme points and two intermediate values (i.e., 0%, 20%, 40% and 60%). As in the previous case, the



(a) Stream vs FFT



(b) Radix vs FFT

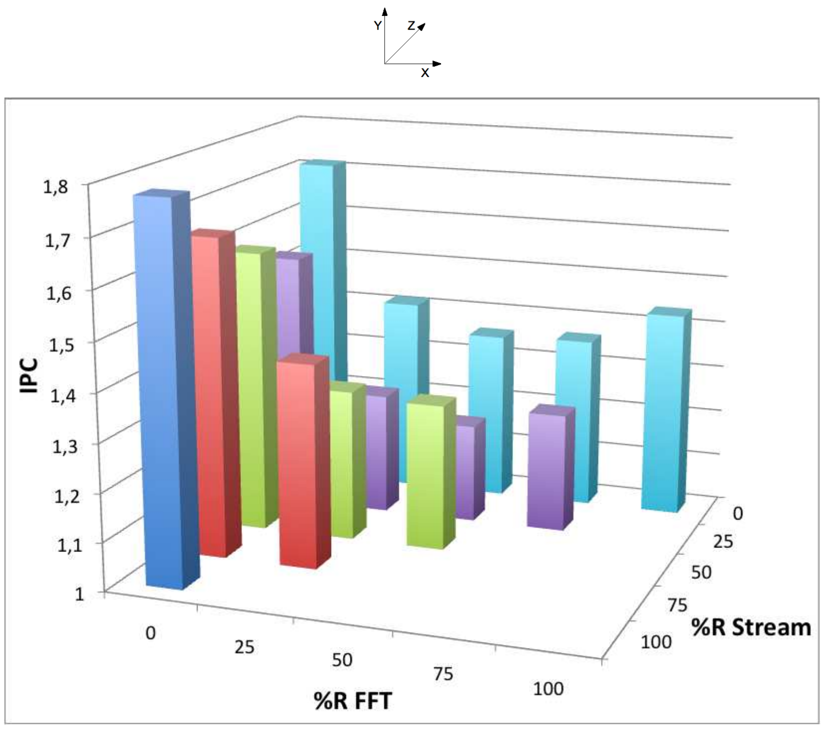
FIGURE 4.5: Overall IPC for different memory assignments assuming a 60% of remote memory usage.

maximum system performance comes at the expense of unacceptable performance for a given application. For instance, in Figure 4.5 (a), the best system performance (about 0.73) is achieved when assigning the remote memory (i.e., $R = 60\%$) only to Stream, thus clearly damaging its performance. Again, the QoS must be considered to avoid the performance degradation.

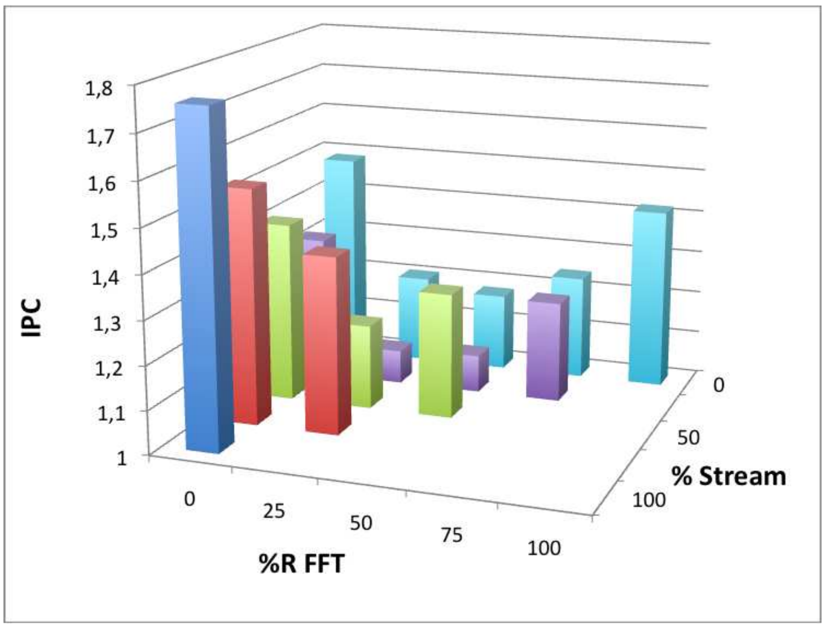
In summary, allocating a fraction of remote memory is analogous to allocating all the remote memory, since the only difference lies on the range of memory that is assigned, which is narrower, thus this is a particular case of the previous one. The same reasoning can be applied to the QoS parameters, where the maximum range of assigned remote memory is also limited in extent.

4.2.3 Extending the Analysis to More Applications

This section extends the analysis for a number of applications higher than two. Four different mixes have been evaluated and represented in Figure 4.6 and Figure 4.7: mix1= {Stream, FFT, Cholesky}, mix2= {Stream, FFT, Radix}, mix3= {Stream, Radix, Cholesky}, and mix4= {FFT, Radix, Cholesky}. The mixing criterion has consisted in making all the possible combinations of the three benchmarks in sets of three elements. In these plots, values of X and Z axes refer to the percentage of remote memory assigned to two of the three applications, and $100\% - (X\% + Z\%)$ corresponds to the remote memory assigned to the third application. The Y-axis stands for the total system performance and its highest point shows the maximum performance achieved in the system. As in the previous analysis, the maximum is reached when all the remote memory is assigned to only one application. This application can be chosen as discussed in Section 4.2.2.1. For example, when running mix1 (see Figure 4.6(a)), the maximum IPC (1.75) is achieved when the whole remote memory is assigned to Stream. For this mix, the maximum performance leads to very poor performance for this application. Again, this fact can be controlled by means of the QoS parameter. For example, by setting the QoS of Stream about 15%, its IPC will not drop below 0.1 (see Figure 4.1(b)). In Figure 4.6(a), this is equivalent to remove the columns that do not fulfill the required QoS for Stream (i.e., from $Z = 25\%$ to $Z = 100\%$).

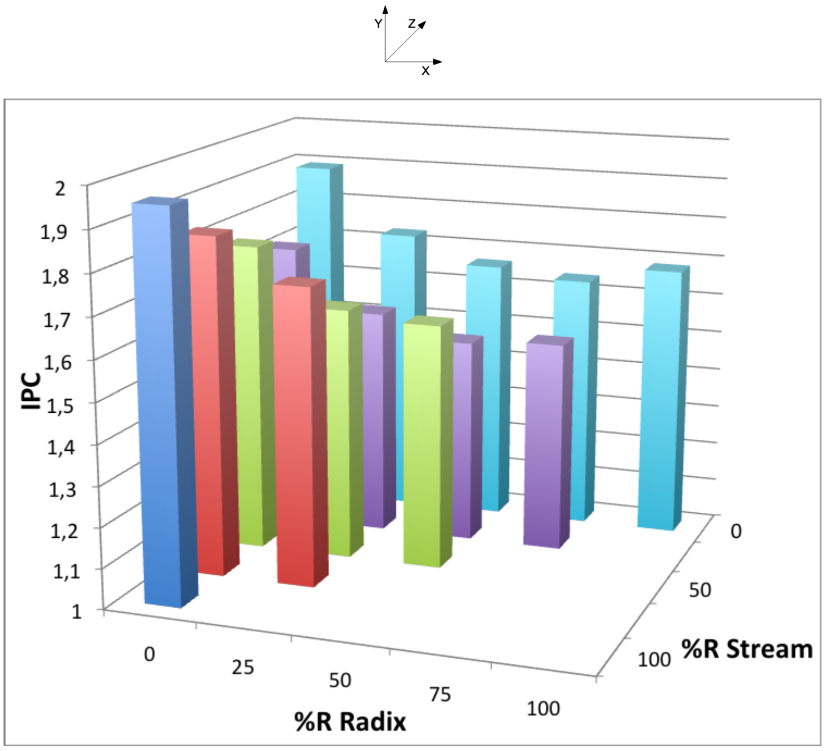


(a) Stream vs FFT vs Cholesky

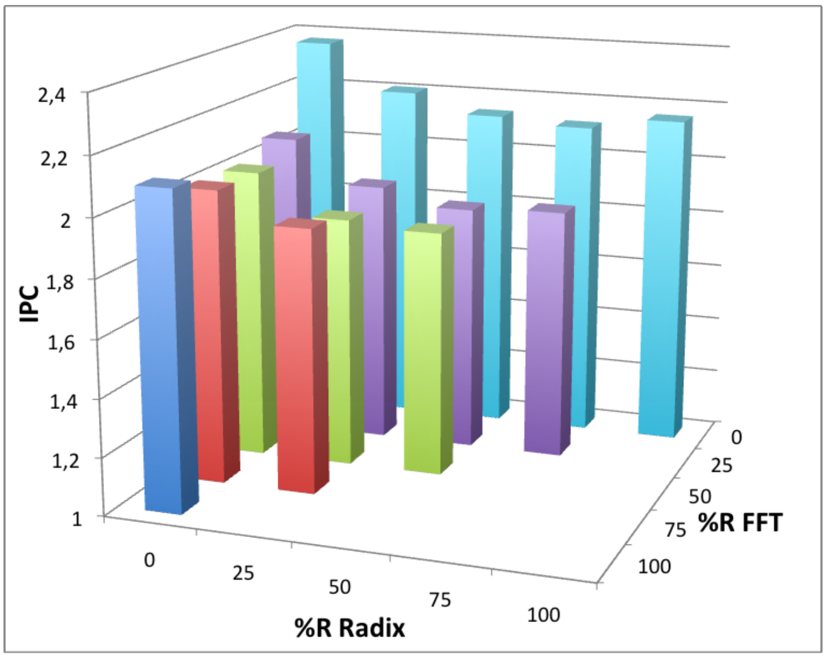


(b) Stream vs FFT vs Radix

FIGURE 4.6: Overall IPC for different memory assignments for three concurrent applications (I).



(a) Stream vs Radix vs Cholesky



(b) FFT vs Radix vs Cholesky

Figure 4.7: Overall IPC for different memory assignments for three concurrent applications (II).

4.3 Proposed Memory Scheduler

This section presents the memory allocation algorithm devised from the previous analysis. The aim of the algorithm is to distribute the available memory in the three memory regions among n applications running on the nodes of a given local board. This work assumes a static approach where the performance of each application has been profiled off-line varying R for a few points. This profile is provided as an input to the algorithm.

For the sake of clarity, the algorithm is split in two main parts: remote memory assignment and local memory assignment. Remote memory can be assigned by two different algorithms: ideal and heuristic. The former one, referred to as SPP (set of possible permutations) provides the best distribution but it requires a high computational cost, as it is based on an exhaustive search. SPP is useful as a reference to identify the maximum achievable IPC. The heuristic algorithm implements a cost-effective heuristic that reduces the computational cost of SPP in a $(n-1)!$ factor while providing memory distributions close to or the same as SPP.

4.3.1 *SPP* Remote Memory Scheduler

Figure 4.8 describes the *SPP* remote memory scheduler. The algorithm first checks if there is a need to use remote memory, that is, if the application requires more memory than the available DRAM in its motherboard (see LABEL 1). On such a case, it makes a search of the optimal remote memory distribution that maximizes the aggregated IPC of all the applications (see LABEL 2). After that, it allocates the remote memory for each application following this distribution (see LABEL 3).

A tuple RM composed of n values $(RM_0, RM_1, ..., RM_{n-1})$ is used to represent a given remote memory distribution among applications, where each value RM_i is the percentage of remote memory assigned to i application. Thus, the sum of the values of a given tuple is 100%. The algorithm has been designed with a QoS parameter to avoid unacceptable performance; that is, it must be fulfilled that the remote memory

```
Algorithm: SPP remote memory scheduler with QoS constraint
Data:
  $n$: number of running applications in the system
  $L$: Available local memory
  $R$: Available remote memory
  $M_i$: Remaining memory required by $i$ application
  $QoS_i$: maximum allowed remote memory for $i$ application
  $P$: Set of all the possible permutations of $n$ integers from 0 to $n-1$
  $IPCest_i(x)$: IPC estimation based on the profiled points of a given memory
  assignment $x$ to app. $i$
  $RM_i$: Percentage of remote memory assigned to $i$ application. Initially, all its
  components are null

LABEL 1: CHECK IF THERE IS ENOUGH MEMORY IN THE
MOTHERBOARD
if $\Sigma M_{i,\forall i=0..n-1} > L$ then

  LABEL 2: FIND THE OPTIMAL REMOTE MEMORY DISTRIBU-
  TION
  $maxIPC \leftarrow 0$
  for all $p \in P$ do
    $permIPC \leftarrow 0$
    $AM \leftarrow 0\%$
    for $j = 0$ to $n-1$ do
      $RM_{p_j} \leftarrow MIN(QoS_{p_j}, 100\% - AM)$
      $permIPC \leftarrow permIPC + IPCest_{p_j}(RM_{p_j})$
      $AM \leftarrow AM + RM_{p_j}$
      if $AM = 100\%$ then
        exit
      end if
    end for
    if $permIPC > maxIPC$ then
      $max \leftarrow RM$
      $maxIPC \leftarrow permIPC$
    end if
  end for

  LABEL 3: ALLOCATE REMOTE MEMORY
  for $i = 0$ to $n-1$ do
    Allocate $RM_i \times R$ in remote motherboard to $i$ application
    $M_i \leftarrow M_i - RM_i \times R$
  end for
end if
```

FIGURE 4.8: SPP algorithm to distribute remote memory among n applications.

assigned to i application must be lower or equal than its QoS_i (i.e., $RM_i \leq QoS_i \; \forall \; i$ application).

The results discussed in Section 4.2 showed that the best performance is achieved when assigning the maximum allowed QoS_i to the i application with the least IPC contribution. The simplest case arises when the remote memory is only distributed between two concurrent applications, A0 and A1. In this case, there are only two choices: assigning as much remote memory as possible to A0 and the remaining to A1, and vice-versa.

For a higher number of applications, the algorithm uses a *priority vector*, where the position of a given application in the vector indicates the order in which remote memory is assigned. For instance, let us consider a system executing three concurrent applications with a priority vector $\vec{p} = (A0, A1, A2)$ and a quality of service vector $\vec{QoS} = (30, 40, 50)$. The priority vector states that the remote memory is first assigned to application A0, then to A1, and finally to application A2. Thus, the assigned remote memory is defined as $\vec{RM} = (30, 40, 30)$. In contrast, for a priority vector $\vec{p} = (A1, A2, A0)$, the algorithm provides $\vec{RM} = (10, 40, 50)$. That is, in the latter case, the remote memory is first assigned to A1, then to A2, and finally to A0.

The set P of possible combinations of the priority vector for n applications matches the set of all the possible permutations of n integers from 0 to $n-1$. For example, for $n = 3$ the set is composed of $P = \{(0,1,2), (0,2,1), (1,0,2), (1,2,0), (2,0,1), (2,1,0)\}$. This set is used by the SPP algorithm and can be obtained with the Steinhaus-Johnson-Trotter algorithm [45] whose computational cost is $n!$

Regarding the profile size, note that when plotting IPC as a function of the assigned remote memory, the curve looks sufficiently defined using five points. Thus, although the curve could be better defined with more points, its potential benefits would not be compensated with the cost of profiling the additional points. Therefore, the IPC profile of the proposed algorithms consists of the values obtained with five (0%, 25%, 50%, 75% and 100%) remote memory assignments for each application.

The algorithm also estimates the IPC of a given memory assignment when it falls in between two profiled values (e.g., 20%). This estimation can be done in two main ways: i) by using the value of the closest profiled point and ii) by using an approximation method. A linear approximation shows a good tradeoff between speed and accuracy since it can be quickly computed (i.e., just a multiplication and a sum operation are required). Of course, more accurate results could be obtained with complex methods like quadratic approximation. Once the optimal remote memory distribution has been found, the remote memory is allocated to applications, and the assigned memory M_i is updated. Finally, the pending memory is assigned to the local board as discussed later (see Section 4.3.3).

4.3.2 Remote Memory Scheduling Heuristic

The heuristic presented in Figure 4.9 provides a remote memory distribution close to the optimal. As in the previous case, the heuristic relies on the profiled values and takes into account that the best memory performance is achieved when assigning the maximum allowed remote memory to the application with the least IPC contribution.

As the SPP algorithm, the heuristic is only applied if there is not enough local memory (see LABEL 1). In this case, it selects the application whose IPC is the least affected by the remote memory assignment, that is, the minimum value between its QoS and the remote memory still pending to be assigned (i.e. $100\% - AM$) in the system (see LABEL 2). Then, the heuristic calculates and assigns the percentage of remote memory ($RM_{penalized}$) that corresponds to the application selected above (see LABEL 3). The process of assigning remote memory to applications continues until the total remote memory required among the running applications has been assigned. Finally, once the corresponding percentages have been obtained, the remote memory is allocated to applications (LABEL 4).

```
Algorithm: Remote memory scheduling heuristic with QoS constraint

  $n$: number of running applications in the system
  $L$: Available local memory
  $R$: Available remote memory
  $M_i$: Remaining memory required by $i$ application
  $QoS_i$: maximum allowed remote memory for $i$ application
  $IPCest_i(x)$: IPC estimation based on the profiled points of a given memory assignment $x$ to app. $i$
  $RM_i$: Percentage of remote memory assigned to $i$ application. Initially, all its components are null

LABEL 1: CHECK IF THERE IS ENOUGH MEMORY IN THE MOTHERBOARD
if $\Sigma M_{i,\forall i=0..n-1} > L$ then

  $AM \leftarrow 0\%$
  while $AM < 100\%$ do
    LABEL 2: FIND THE APPLICATION LEAST AFFECTED BY REMOTE MEMORY
    $minimpact \leftarrow \infty$
    for $i = 0$ to $n - 1$ do
      if $RM_i = 0\%$ then
        $assig \leftarrow MIN(QoS_i, 100\% - AM)$
        $impact \leftarrow IPCest_i(0) - IPCest_i(assig)$
        if $impact < minimpact$ then
          $minimpact \leftarrow impact$
          $penalized \leftarrow i$
        end if
      end if
    end for
    LABEL 3: REMOTE MEMORY ASSIGNMENT TO THE LEAST AFFECTED APP.
    $RM_{penalized} \leftarrow MIN(QoS_{penalized}, 100\% - AM)$
    $AM \leftarrow AM + RM_{penalized}$
  end while

  LABEL 4: ALLOCATE REMOTE MEMORY
  for $i = 0$ to $n - 1$ do
    Allocate $RM_i \times R$ in remote motherboard to process i
    $M_i \leftarrow M_i - RM_i \times R$
  end for
end if
```

FIGURE 4.9: Heuristic to distribute remote memory among n applications.

4.3.2.1 Working Examples

Let us discuss how the heuristic performs through two working examples: i) no application has any QoS constraint, and ii) the applications have QoS requirements. Each working example analyzes four cases or mixes (see Section 4.2.3), each one composed of three benchmarks.

The simplest example arises when the algorithm works with no QoS constraint, that is, all the applications have their QoS parameter equal to 100%. Table 4.3 shows the mixes and how the heuristic solves each particular case (see column *Assigned R*). In this example, the overall IPC corresponds to the value of the highest bar of each graph illustrated in Figure 4.6 and Figure 4.7.

When the applications have QoS requirements and the sum of these values is equal to 100%, the only thing that the heuristic has to do is assigning a percentage of remote memory equal to its QoS to each application. On the other hand, when the sum of the QoS values is greater than 100%, at least one application will receive an amount of remote memory less than its QoS.

The second working example focuses in the latter behavior. Table 4.4 shows the QoS of the mixes and how the heuristic solves each case. Notice that the QoS vector

TABLE 4.3: Working example without QoS constraints: $\vec{QoS} = (100, 100, 100)$.

Mix	Applications	Quality of Service(%)	Assigned R(%)	Overall IPC
1	Stream	100	100	1.773
	FFT	100	0	
	Cholesky	100	0	
2	Stream	100	100	1.756
	FFT	100	0	
	Radix	100	0	
3	Stream	100	100	1.952
	Radix	100	0	
	Cholesky	100	0	
4	FFT	100	0	2.371
	Radix	100	0	
	Cholesky	100	100	

TABLE 4.4: Working example with QoS restrictions: $\vec{QoS} = (20, 55, 70)$.

Mix	Applications	Quality of Service(%)	Assigned R(%)	Overall IPC
1	Stream	20	20	1.542
	FFT	55	10	
	Cholesky	70	70	
2	Stream	20	20	1.332
	FFT	55	10	
	Radix	70	70	
3	Stream	20	20	1.721
	Radix	55	10	
	Cholesky	70	70	
4	FFT	20	0	2.207
	Radix	55	30	
	Cholesky	70	70	

in the four cases is $\vec{QoS} = (20, 55, 70)$, so the sum of its components is not only greater than 100% but also these values do not correspond to any profiled point, so the algorithm must estimate them. To this end, it has been assumed that the algorithm approximates to the closest profiled point. For the first three mixes, the best remote memory distribution vector is $\vec{RM} = (20, 10, 70)$. The estimated overall IPC for these situations is approximated to the profiled remote memory percentages 25%, 0% and 75%, respectively. For mix 4, the heuristic provides a memory distribution of $\vec{RM} = (0, 30, 70)$, whose overall IPC is estimated by means of the values of the profiled points 0%, 25% and 75%, respectively.

4.3.2.2 Cost Analysis

The SPP algorithm carries out a thorough search among the set of the possible remote memory assignments (i.e., $n!$) to find out the combination that optimizes the overall IPC. This set grows factorially with the number of applications running on the system so it leads to prohibitive computational costs for large sets. On the contrary, the devised heuristic algorithm finds an optimal or near-optimal remote memory distribution but largely reduces the computational cost by performing a reduced search.

Table 4.5: Computational cost comparison.

Algorithm	**# Cases**	**Cost/case**	**Total cost**
SPP	$n!$	n	$n \times n!$
Heuristic	n	n	$n \times n$

Table 4.5 shows the computational costs of both schedulers. The SPP algorithm iterates through $n!$ different cases of n-cost each. As a result of the limitation in the number of possible remote memory assignments explored, the proposed heuristic reduces the number of analyzed cases from $n!$ to n, thus noticeably improving the total computational cost of the SPP algorithm.

To sum up, the proposed heuristic reduces the computational cost in a factor of $(n-1)!$ while providing reasonable performance since all the QoS of the applications are satisfied. On the contrary, the SPP algorithm might provide better performance for some mixes but at the expense of a much higher computational cost.

4.3.3 Local Memory Assignment

Once the remote memory has been scheduled to applications by one of the studied schedulers, the local memory must be assigned to complete the memory allocation. Figure 4.10 shows the local memory scheduler. For each application and following a circular order, the scheduler looks for free memory in each node of the local motherboard, beginning by the node where the application is running on. This process goes on until the required memory has been completely assigned to applications.

4.4 Summary

This chapter has presented an ideal algorithm and a heuristic strategy to assign main memory, which can be located in three main regions (local to node, local to board, or remote), among applications running on a RMA-interconnected cluster. With this

```
Algorithm: Local memory scheduler

Data:
  $n$: number of running applications in the system
  $M_i$: Remaining memory required by $i$ application
  $L_i$: Memory available in node $i$
for $i = 0$ to $n - 1$ do
  if $M_i > 0\%$ then
    $j \leftarrow i$
    repeat
      $toAlloc \leftarrow MIN(M_i, L_j)$
      $M_i \leftarrow M_i - toAlloc$
      $L_j \leftarrow L_j - toAlloc$
      $j \leftarrow (j + 1) MOD(n)$
    until $M_i = 0\%$
  end if
end for
```

FIGURE 4.10: Algorithm to distribute local memory among n applications.

aim, three main steps have been followed to design these schedulers. For each one, different conclusions can be drawn.

First, since benchmarks have different memory requirements, the impact on performance of each application when varying the memory distribution among regions (L, Lb, and R) has been studied. This study has shown that, i) the memory distribution between L and Lb can impact on performance when no R memory is allocated, ii) the previous distribution has a slight effect on performance when R memory is allocated, and iii) performance degradation widely varies among applications when allocating R memory.

Second, the system performance when several benchmarks running concurrently compete for memory, has been analyzed. This study has shown that the total performance is benefited when R memory is assigned (as much as possible) first to the application that least degrades its performance, then to the second one, and so on. However, this assignment strategy can lead to unacceptable performance degradation for some applications. Hence, a quality of service parameter has been defined for each application.

From these studies, two memory scheduling algorithms have been presented. SSP, which is an ideal algorithm that makes a search of the optimal memory distribution for all the applications, and a heuristic strategy that approximates the best local and remote memory distribution among applications has been presented. The heuristic has been designed to guarantee a minimum QoS performance for each benchmark while optimizing the global system performance. Results have shown that the memory distribution provided by the heuristic is close or the same as the optimal distribution found by the ideal algorithm (SPP), whose computational cost is prohibitive for a high number of applications.

A summary of the work presented in this chapter has been published in [46].

Chapter 5

Scheduling Policy based on a Performance Predictor

In the previous chapters we have discussed the importance of properly balancing the main memory resources among the applications running in cluster computers. In particular, the impact on performance may be significant when RMA hardware is used to extend the addressable memory space of selected motherboards to the memory installed in a remote motherboard of the same cluster.

In this context, the memory scheduler is in charge of finding a suitable distribution of local and remote memory that maximizes the performance and guarantees a target QoS among the applications. Changing the memory distribution is a slow process that involves several motherboards, thus the memory scheduler needs to make sure that a new distribution provides better performance than the current one.

To estimate the expected performance of a given memory distribution, the memory schedulers described in Chapter 4 performed an off-line profiling of the benchmarks for different memory distributions. On the contrary, the scheduling solution presented in this chapter dynamically predicts at run-time the performance of the benchmarks by measuring their utilization of the system resources during the execution. The proposed predictor is driven by a novel performance model fed by simple hardware counters, which are typically available in most current processors, that measures the distribution of the execution time devoted to the processor, memory, and network resources.

5.1 Performance Model

A system whose running applications can be executed with different memory distributions (L, Lb, R) needs a mechanism to determine which memory distribution should be assigned to each application. This section presents a methodology for predicting the impact on performance of the different memory distributions, and then, the predictions are used to guide the assignment of memory regions to applications in order to meet memory constraints and achieve the best performance.

This work assumes that the predictor evaluates a set of possible memory distributions. In particular, for experimental purposes we evaluated seven distributions (three samples and four estimated cases) since this number of data points is enough to define the performance of each application among the complete set of possible memory distributions [46]. An analytical method is proposed to predict the performance (execution time) of a running application A when having a memory assignment $\{L = X,\ Lb = Y,\ R = Z\}$.

The modeled application-to-memory assignment prediction mechanism makes use of the performance counters available in current processors to track the number of cycles spent by each considered event during a full scheduling quantum.

5.1.1 Analytical Model

The execution time of a given application can be estimated from two main components, as stated by equation 5.1.

$$T_{ex} = C_{Dispatch} + C_{mem\_stalls} \tag{5.1}$$

Each C_x is the number of processor cycles spent on x activity. As the dispatch width has been assumed to be 1, the execution time can be expressed as the sum of the number of dispatched instructions (i.e., $C_{Dispatch}$) plus the number of stall cycles due to memory accesses (i.e., C_{mem_stalls}). This simplified model assumes that no stall due to branch misprediction arises. In order to relax the huge memory latency due to an access to a remote motherboard, memory instructions are allowed to execute in parallel. That is, a look-up free cache has been assumed. The cache considers a 8-entry MSHR (Miss Status Hold Register).

In the devised model, stalls due to a lack of free entries in the load-store queue (LSQ) will become critical for performance, mainly in those benchmarks having a high rate of memory accesses. On the other hand, the dispatch stage will remain stalled during the execution of a load instruction in the case of in-order execution. This includes both the accesses to caches (e.g. L2 caches) and to the main memory, with their respective access times as well as the delays related to the network or structural hazards.

To project the IPC, the performance model breaks down the memory components of the execution time into memory region-dependent (i.e., C_L, C_{Lb}, C_R) and memory region-independent components (i.e., C_{caches}):

$$C_{mem\_stalls} = C_L + C_{Lb} + C_R + C_{caches} \tag{5.2}$$

C_L, C_{Lb}, and C_R refer to the number of cycles spent on each memory region, that is, Local, Local to Board, and Remote. Each $C_{<region>}$ includes the cycles spent by the operations accessing to this memory region.

In particular, stalls due to the main memory region and caches access time have been taken into account.

The Main memory access time includes both the cycles spent in the data read from main memory and the message transfer through the memory network.

The Caches access time considers L2 accesses and stalls due to L1 accesses and dependent instructions. During these kind of accesses, no memory module is accessed, thus having the same value regardless the memory distribution. Then *Caches access time* is considered region-independent.

The final equation used by the performance predictor is 5.3:

$$T_{ex} = C_{Dispatch} + C_L + C_{Lb} + C_R + C_{caches} \tag{5.3}$$

5.1.2 Estimating Performance

The model assumes that the implemented target machine provides the required performance counters to obtain the values for the variables of equation 5.3. Notice that network traffic is taken into account, so congestion is also quantified.

The predictor requires to run each benchmark three times to gather the required values to project the performance. Each sample is associated with the execution time, i.e., T_{ex}, obtained when all the memory accesses are performed in one single region, that is, *i)* all the accesses to local memory region (i.e., $T_{ex,L=100\%}$), *ii)* all the accesses to the other node in the local motherboard memory region (i.e., $T_{ex,Lb=100\%}$), and *iii)* all the accesses to remote memory region (i.e., $T_{ex,R=100\%}$):

Sample 1 (L = 100%, Lb = 0%, R = 0%): $T_{ex,L=100\%} = C_{L(L=100\%)} + C_{caches}$

Sample 2 (L = 0%, Lb = 100%, R = 0%): $T_{ex,Lb=100\%} = C_{Lb(Lb=100\%)} + C_{caches}$

Sample 3 (L = 0%, Lb = 0%, R = 100%): $T_{ex,R=100\%} = C_{R(R=100\%)} + C_{caches}$

To predict the execution time for a given memory distribution, the predictor calculates a weighted execution time, $T_{ex\_estimated}$, from the three samples. It takes each not null memory region component $C_{<region>}$ of each of the samples and multiplies it by the fraction $f_{<region>}$ of accesses of the destination memory region:

$$T_{ex\_estimated} = C_{L(L=100\%)} \cdot (f_L) + C_{Lb(Lb=100\%)} \cdot (f_{Lb}) + C_{R(R=100\%)} \cdot (f_R) + C_{caches} \tag{5.4}$$

For any given memory distribution, equation 5.4 can be used to predict its execution time given the gathered components for the three samples. This provides a mechanism to identify the optimal memory distribution to run a given execution phase with minimal performance loss. So this prediction will be an input for the memory scheduler.

Table 5.1 shows an example for FFT. The first column $C_{<region>}$ provides the time gathered for the three samples. Then using these values, the execution time for the memory distribution $(50\%, 50\%, 0)$ is estimated as specified in column $f_{<region>}$. The contribution of each region to the execution time is shown in the last column $C_{<region>\_weighted}$. The last row shows the total estimated execution time as 2774807.8 cycles. To verify the accuracy of the model, we compared its estimate against the time obtained from the detailed simulation that was 2774931. Thus the model has obtained an estimate which deviates less than 0.005% from the target value.

Table 5.1: Performance predictor working example.

	$C_{<region>}$	$f_{<region>}$	$C_{<region>\_weighted}$
Sample1	44687	0.5	22343.5
Sample2	62236	0.5	31118
Sample3	166757	0	0
C_{caches}			2721346.3
$t_{ex\_estimated}$			2774807.8

5.2 Validating the Model

This section analyzes the prediction accuracy. Different experiments were launched with mixes consisting of four benchmarks (FFT, Cholesky, Radix, and Stream) for the following eight memory distributions: *i)*(100%, 0%, 0%), *ii)*(50%, 50%, 0%), *iii)*(0%, 100%, 0%), *iv)*(75%, 0%, 25%), *v)*(50%, 25%, 25%), *vi)*(50%, 0%, 50%), *vii)*(25%, 0%, 75%), *viii)*(0%, 0%, 100%). Then, we have taken the components of the three samples (*i*, *iii*, and *viii*) and applied the model to each benchmark to obtain the execution time for each remaining memory distributions. Finally, the Instructions Per Cycle (IPC) has been calculated for each case.

The devised predictor has been validated by comparing its estimates against the performance results obtained by the execution of the benchmarks.

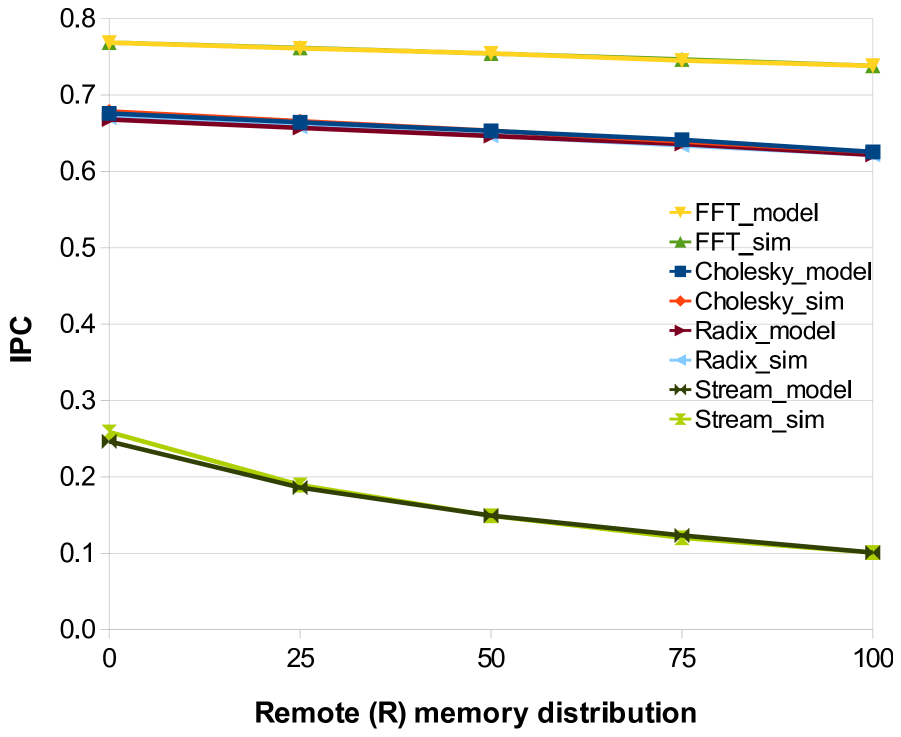


FIGURE 5.1: Model Validation. Detailed cycle-by-cycle simulation vs model.

Figure 5.1 shows the comparison of the simulated performance results (*sim*) against the values calculated by the performance predictor (*model*). Both model and detailed cycle-by-cycle simulation curves are overlapped, because the model provides a deviation lower than 5% in the worst case, being near to 0% for some of the benchmarks, for instance, FFT.

5.3 Summary

This chapter has presented a performance predictor which is able to estimate the execution time for a given memory distribution of an application. We first carried out a study to determine the events considered by our model, and classified them as memory-region dependent and independent. The model assumes that the number of cycles spent in each considered event is obtained from some hardware counters of the target machine.

The performance predictor provides the input to a memory scheduling mechanism which, fed by the estimated performance values, is able to dynamically choose the optimum target memory distribution for each application concurrently running in the system in order to achieve the best overall performance.

The validation study shows that the dynamic predictor is very accurate, since its deviation from the real results is always lower than 5% and very close to 0% in several studied cases.

The main conclusions and results if the work discussed in this chapter has been published in [47].

Chapter 6

Scheduling Policy based on page granularity

Cluster systems typically support interleaved memory at cache-block granularity. As the operating system manages memory at page level granularity, this chapter studies the impact on the system performance when working at this granularity to distribute memory pages among the memory regions (i.e., local or remote). This behavior is analyzed and compared to a typical interleaved memory distribution at cache block level granularity.

Experimental results show that simply supporting interleaved memory at OS page granularity is a feasible solution that does not impact on the performance of most of the benchmarks. Based on this observation we have investigated the reasons of performance drops in those benchmarks showing unacceptable performance when working at page granularity.

The results of this analysis have led us to propose two memory allocation policies, namely on-demand (OD) and Most-accessed in-local (Mail). The OD policy first places the requested pages in local memory, once this memory region is full, the subsequent memory pages are placed in remote memory. This policy shows good performance when the most accessed pages are requested and allocated before than the least accessed ones, which as proven in this work, is the most common case. Nevertheless, this strategy has poor performance when a noticeable amount of the least accessed pages are requested before than the most accessed ones. This performance drawback is solved by the Mail allocation policy by using profile information to guide the allocation of new pages.

This chapter also characterizes the behavior of the entire SPEC CPU2006 benchmark suite [37] under several memory allocation schemes. A set of metrics such as the execution time, misses per kilo-instruction, and the distribution of L1 accesses are analyzed in order to provide a sound understanding of the effects of the memory behavior in the system performance.

The analysis allows us to classify the applications based on their behavior for a given memory distribution. The different memory distributions affect the execution time of a given application in different ways; thus, it is important that the scheduler is able to estimate how a given memory distribution impacts on performance. In this way, this knowledge could prevent the memory scheduler from choosing a memory distribution that could damage not only the performance of an application but also the overall system performance.

6.1 Memory Allocation Granularity and Memory Interleaving

This chapter focuses on memory allocation policies working at OS page granularity. For performance comparison purposes we have modeled a typical system implementing interleaved memory at cache block granularity. In addition, since the proposed schemes work at page granularity, we have modeled a page-level interleaved scheme to check how sensitive the applications are to the granularity size of the interleaved schemes. Below we discuss the interleaved schemes.

- **Block-level interleaved.** This scheme referred to as *BI* assumes that cache blocks are allocated to local and remote memory in an interleaved way (e.g. even blocks are in local memory and odd cache blocks in remote memory). This scheme has been assumed as baseline since it is the typically implemented in current systems.
- **Page-level interleaved.** This scheme, namely *PI*, also allocates memory in an interleaved way but at OS page granularity. The idea behind this scheme is to explore if performance can be acceptable in most benchmarks when working at this coarse granularity. This scheme has been also considered as baseline, since the proposed strategies work at the same granularity.

6.2 Proposed Page-granularity Policies

Next we discuss the proposed page-granularity memory allocation policies, namely on-demand and Most-accessed in-local scheduler. All the policies that work at page level use the virtual memory subsystem to assign a given memory page to one of the two memory regions. That is, whenever a new virtual page is allocated, its physical frame is set depending on which region is chosen by the specific memory allocation policy.

- **On-demand.** This proposal, referred to as *OD*, allocates memory at OS page granularity. It starts allocating pages in local memory, and when the requested pages exceed the local memory capacity, it allocates pages in remote memory which works as an extension of the local memory. Since local and remote memories are both considered as main memory, no swap is performed between both regions in this scheme. In other words, a cache miss whose requested block is found in remote memory is not handled as a page fault by the OS.

- **Most-accessed in-local.** This scheme will be referred to as *Mail.* This proposal tries to improve the performance of the OD scheme by determining which pages should be allocated in local memory. The Mail scheme works as a scheduler that places those pages which are responsible for more cache misses in local memory and the remaining ones in remote memory. In this chapter, we analyze the performance benefits of this scheme when working in a static way.

6.3 Performance Analysis of the Interleaved Memory Schemes

This section explores how the granularity size can impact on the system performance. Figure 6.1(a) shows, for each benchmark, the normalized execution time of the PI policy with respect to the BI scheme.

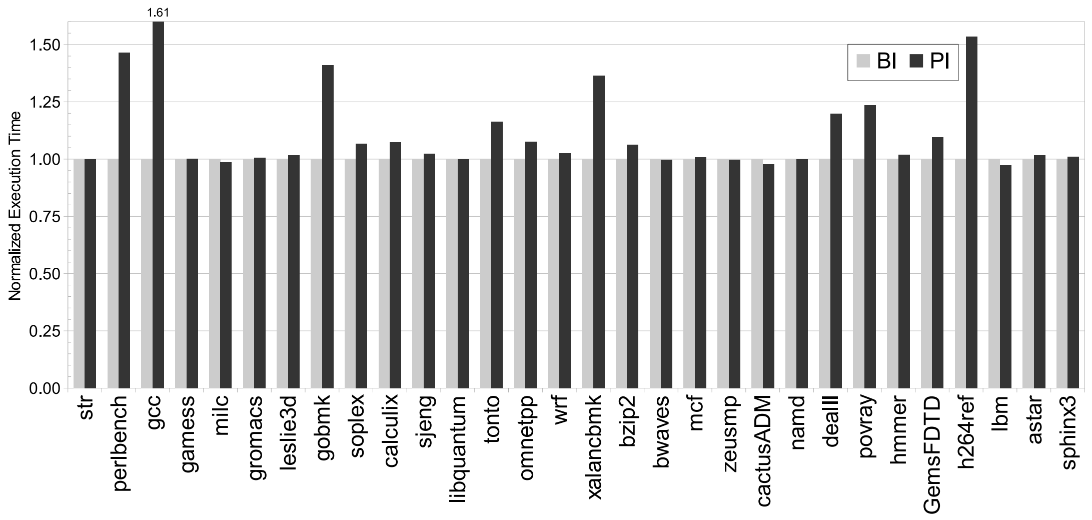
This plot shows the sensitiveness of each benchmark to the granularity of the interleaving. The wider the differences, the more sensitive a given benchmark is. Results show that the execution time of more than half of the benchmarks is not or is scarcely affected when working at page granularity while the execution time of some of them (about one third) grows when working at such large granularity.

The performance penalty widely differs among those benchmarks penalized by large granularities. According to this performance degradation benchmarks can be classified as high degradation, medium degradation and low degradation. For instance, we can include in high degradation (i.e., between 50% and 75%) the *gcc* and *h264ref* benchmarks, in medium degradation (i.e., between 25% and 50%) the *perlbench*, *gobmk*, and *xalankbmk* benchmarks, and in low degradation (i.e., between 2% and 25%) benchmarks like *soplex*, *calculix*, *tonto*, *omnetpp*, *bzip2*, *dealII*, *povray*, and *GemsFDTD.*

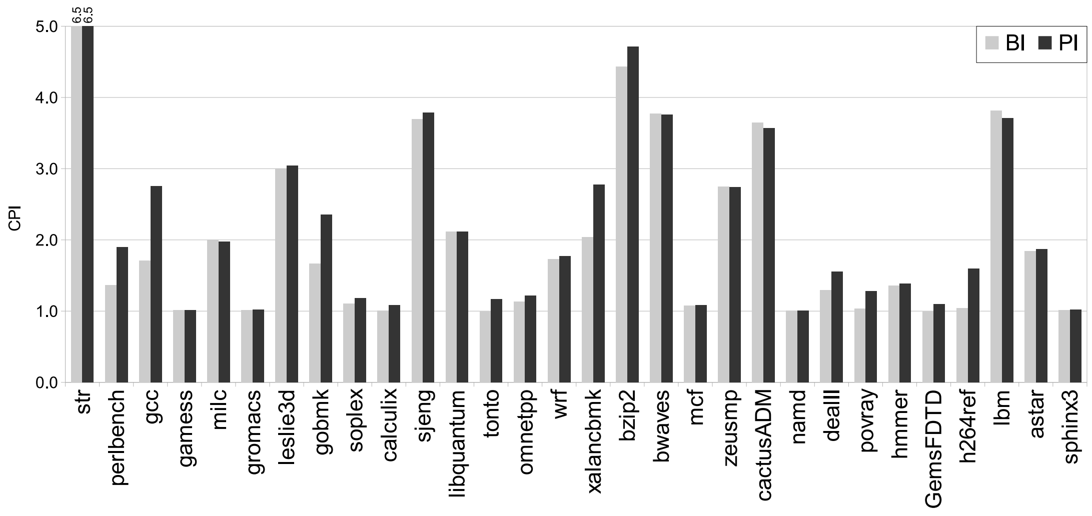
As observed in Figure 6.1(b), benchmarks in the latter group present good performance regardless the interleaving size, since all of them except *bzip2* achieve a Cycles per Instruction (CPI) value close to 1. Notice that a CPI value close to 1 represents a near optimal performance, as we have modeled a single-issue processor.

Since a perfect branch predictor is considered, performance drops mainly come from the memory subsystem. To analyze the memory subsystem behavior of both interleaving schemes we explored the cache hierarchy. Figure 6.2(a) shows, for each benchmark and interleaving, the distribution of load instructions (memory read requests) accesses to the L1 data cache. The results of such loads are classified in *hits*, *delayed hits* (i.e., a hit in a block that is being fetched either from L2 or main memory), and *misses*. We differentiate hits from delayed hits since the latter present a variable latency that may range from main memory latency to a latency close to the one of a conventional hit. Notice that all the benchmarks whose CPI value is below 1.5 present a hit ratio (without considering delayed hits) greater than 0.9.

It can be observed that working at page granularity (PI scheme) increases the number of delayed hits with respect to the BI scheme in some of the benchmarks. The longer the latency of these additional delayed hits, the stronger the impact on performance. Since remote memory accesses present the longest latencies, they have potentially greater impact on the performance. To analyze this fact, we measured those Read Misses Per Kilo-Instruction (RMPKI) in L2 (L2 is the last level cache so a L2 miss incurs a main memory access) that access to Remote Memory (L2 $RMPKI_{RM}$). Figure 6.2(b) shows that the benchmarks whose PI performance degradation (see Figure 6.1(a)) is higher than 25% (i.e., *gcc*, *h264ref*, *perlbench*, *gobmk*, and *xalankbmk*) are those which present a larger L2 $RMPKI_{RM}$ increase with respect to BI. On the other hand, there are some benchmarks such as *str*, *gamess*, *gromacs*, *libquantum*, *zeusmp*, *namd*, and *sphinx3* whose L2 $RMPKI_{RM}$ is below 0.01 for both interleavings. Thus, the impact of the memory interleaving scheme on these benchmarks is negligible. Finally, note that although the L2 $RMPKI_{RM}$ of some benchmarks like *tonto*, *omnetpp*, *dealII*, and *povray* also noticeably grows, their CPI and hit ratio values (see Figure 6.1(b) and Figure 6.2(a)) compensate the performance impact of the growing L2 $RMPKI_{RM}$.

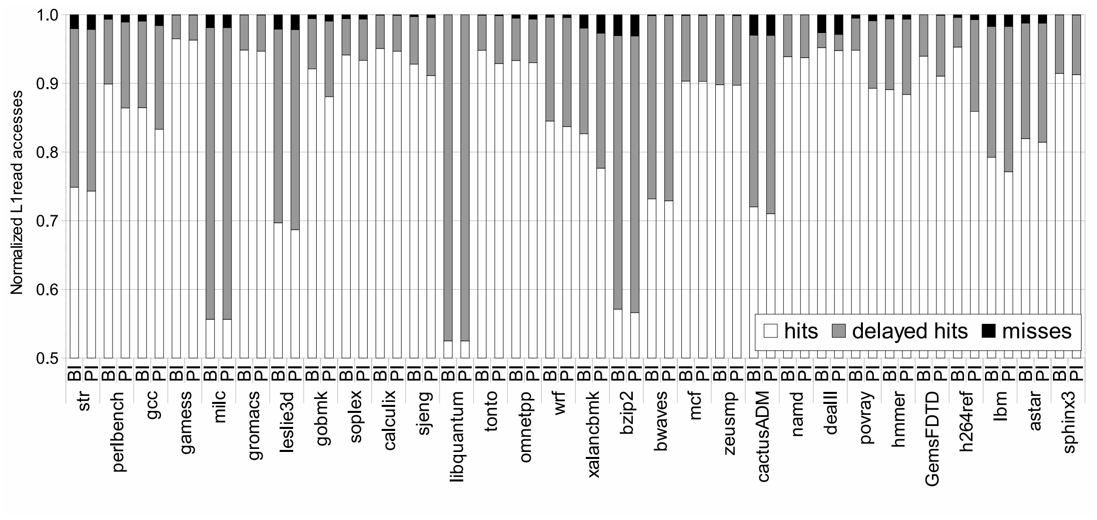


(a) Normalized execution time

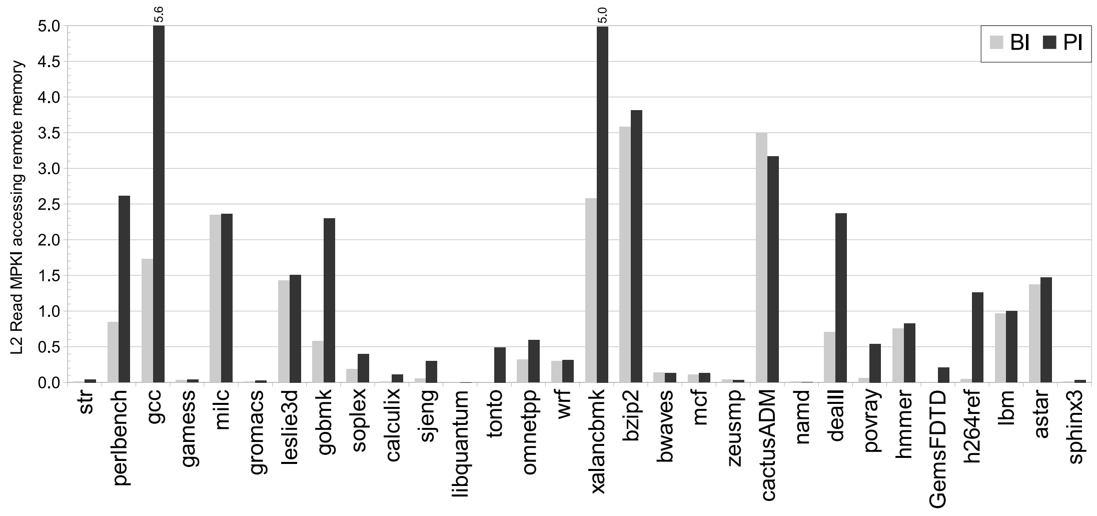


(b) Cycles per instruction

FIGURE 6.1: Interleaved memory configuration.
Performance metrics.



(a) Distribution of L1 cache read accesses



(b) L2 read misses per kilo-instruction accessing remote memory

Figure 6.2: Interleaved memory configuration.
Memory subsystem behavior.

In summary, with respect to the BI scheme, the PI impact on performance is noticeable only for a few benchmarks. The main reason of this performance drop is the increase in the number of L1 delayed hits that must wait until the block that is being fetched (delayed hit) comes from remote memory.

6.4 On-demand Memory Allocation

To avoid the increase of L2 misses accessing to remote memory, we have devised the OD or On-demand memory allocation policy, which places the first accessed pages in local memory as discussed in Section 6.1. This scheme is based on the assumption that the pages that are accessed first will be likely accessed during the whole execution. Thus, if these pages are placed in local memory, the number of accesses to remote memory will be noticeably reduced.

This section analyzes the performance of the OD policy compared to the BI memory organization scheme. The BI scheme assumes, by design, that a half of the working set is allocated to local memory and the other half to remote memory. Therefore, for fair comparison purposes, the devised OD scheme also implements this assumption. In this way, the working set allocated to each memory region is roughly the same as the baseline scheme. For the on-demand policy, this means that the local memory is full when 50% of the accessed pages are brought into memory. From now on, we will refer to this scheme as on-demand 50 or simply OD-50.

Figure 6.3 shows, for each benchmark, the normalized execution time of the OD-50 policy with respect to the BI scheme. Note that an on-demand distribution does not only avoid the performance problems of page interleaving, but also improves the performance achieved by BI in 7 benchmarks.

Performance improvements reach values up to 25% in some cases (*bzip2*, *cactusADM*, and *astar*). The only exception showing significant worse performance for the OD-50 policy is the *hmmer* benchmark.

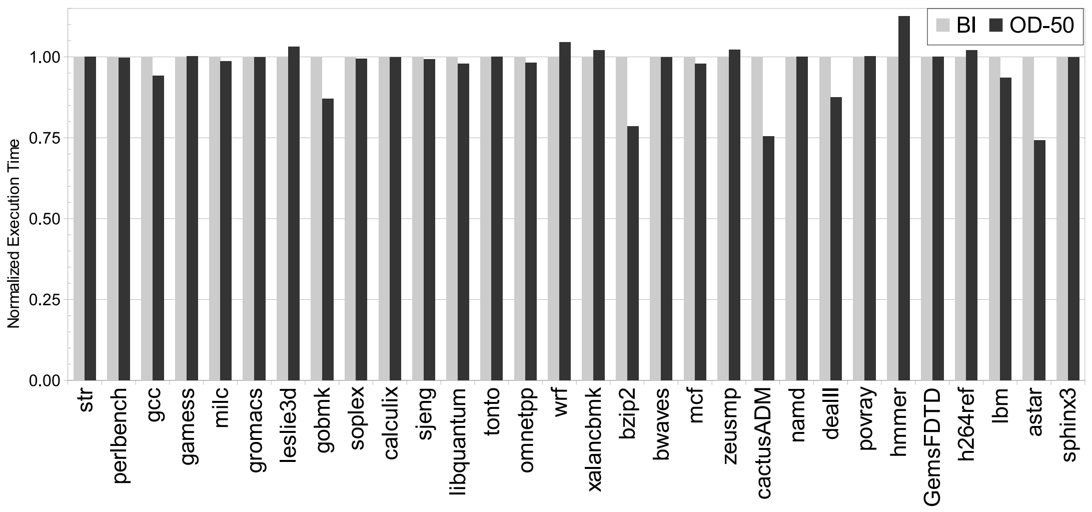
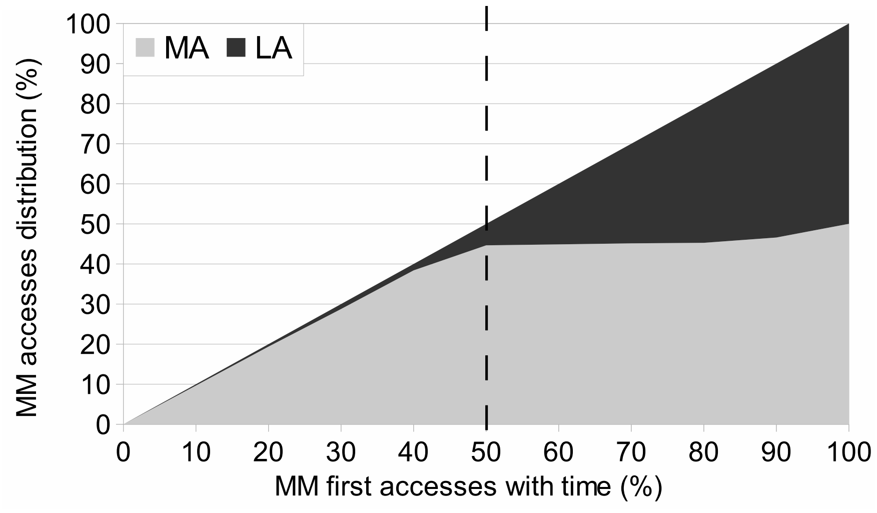


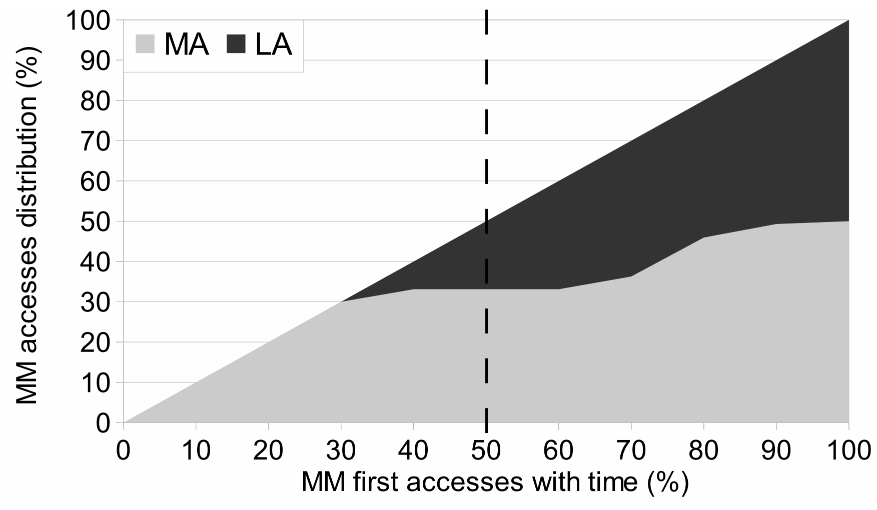
FIGURE 6.3: Block interleave versus On-demand.

We explored the reason behind the variations of OD-50 performance results across the different benchmarks by studying the characteristics of the pages that are allocated to local or remote memory as the execution time advances. The impact of each page on performance mainly depends on its temporal locality quantified as the number of times that it is accessed (i.e., L2 misses accessing that page) and its latency (if it is local or remote). For illustrative purposes, we analyzed this behavior and plotted the results for two of the best performing benchmarks with the OD-50 policy (*astar* and *bzip2*) and another two benchmarks which do not take advantage from this allocation strategy (*hmmer* and *wrf*).

To check how applications take advantage of temporal locality in the OD-50 scheme, pages are classified as most accessed or MA (those 50% of pages which are responsible of more L2 misses) and least accessed or LA (the remaining ones). Figure 6.4 and Figure 6.5 plot the accumulated distribution of MA and LA memory pages as they are brought into memory by the OD-50 scheme. To plot the results only the first access to a given page was considered. Each of the 10 steps in the X axis are equally large in the number of L2 misses (i.e., main memory accesses) but not in time. Depending on whether the local memory is full or not, the OD-50 scheme places that page in remote or local memory, respectively. Notice that this decision only considers the first access to that page since no page replacement is performed.

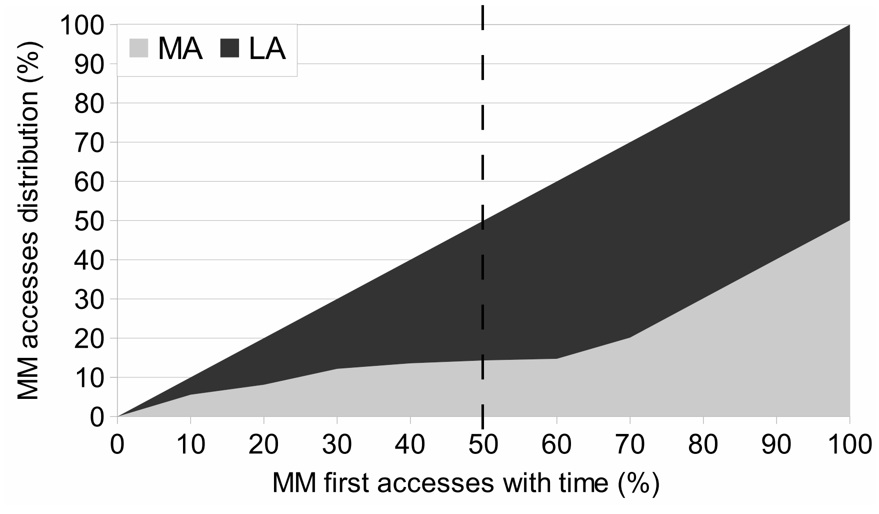


(a) astar

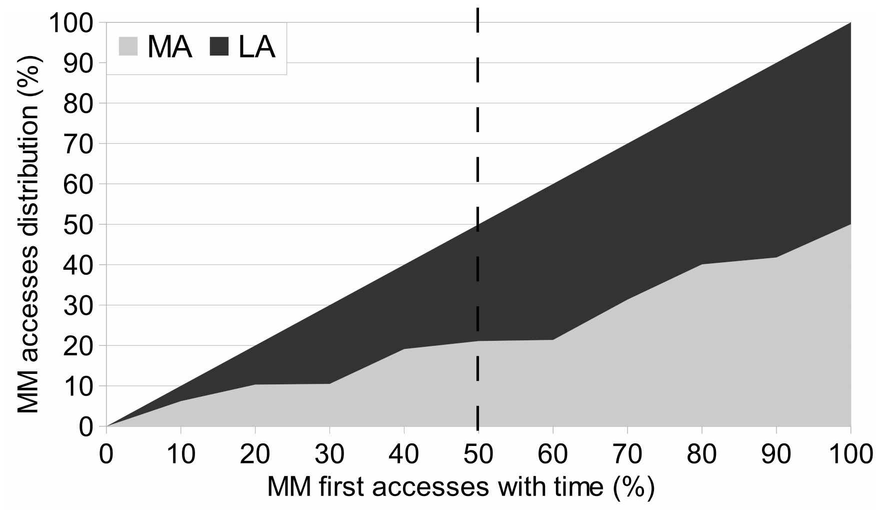


(b) bzip2

FIGURE 6.4: Local and Remote MM first accesses distribution along time. The best performing benchmarks under OD.



(a) hmmer



(b) wrf

FIGURE 6.5: Local and Remote MM first accesses distribution along time. The worst performing benchmarks under OD.

Since both memory regions have the same size, the dashed lines in Figures 6.4 and 6.5 represent the situation where the local memory is full, so subsequent allocated pages will be placed in remote memory. According to temporal locality and OD-50, the more MA pages are allocated in local memory sooner, the better the performance. The ideal situation would happen when all the MA pages are allocated before the dashed line in local memory and all the LA pages after the dashed line in remote memory. In other words, the MA curve grows until the dashed line and then remains constant; from then on, the LA curve starts to grow. As observed, *astar*'s plot (Figure 6.4(a)) resembles the described ideal situation. This can be corroborated by looking at Figure 6.3 which shows that this benchmark reaches the best on-demand performance compared to BI. *Bzip2*'s (see Figure 6.4(a)) distribution also presents a similar plot, but some LA pages are allocated to local memory, so the on-demand distribution can be improved. In contrast, for *hmmer* and *wrf*, which present the worst performance, the distribution shows a high number of LA page allocations before the dashed line (Figures 6.5(a) and 6.5(b)), thus a lot of MA pages are allocated to remote memory, so damaging the performance.

6.5 Mail Memory Allocation

As observed in Figure 6.4 and Figure 6.5, the memory pages distribution obtained by the on-demand policy can be improved even for some of the best performing benchmarks (e.g., *bzip2*). The key is to place the memory pages with the highest number of main memory accesses in the local region. To this end, we devised the Most-accessed in-local (Mail) allocation policy. With the aim of exploring the impact on performance of the Mail policy, profiling information is used, to discern MA from LA pages.

Figure 6.6 shows the normalized execution time of Mail with respect to the OD policy. As in previous experiments, we assume that the memory is split in two equally sized regions. As expected, there are several benchmarks whose performance is improved by Mail. In some cases, this improvement reaches 25% (*xalancbmk*, *bzip2*, and *hmmer*). The fact that those benchmarks presenting and on-demand distribution close to the optimal like *astar* (see Figure 6.4(a)), take scarce benefit from the Mail policy was also foreseeable.

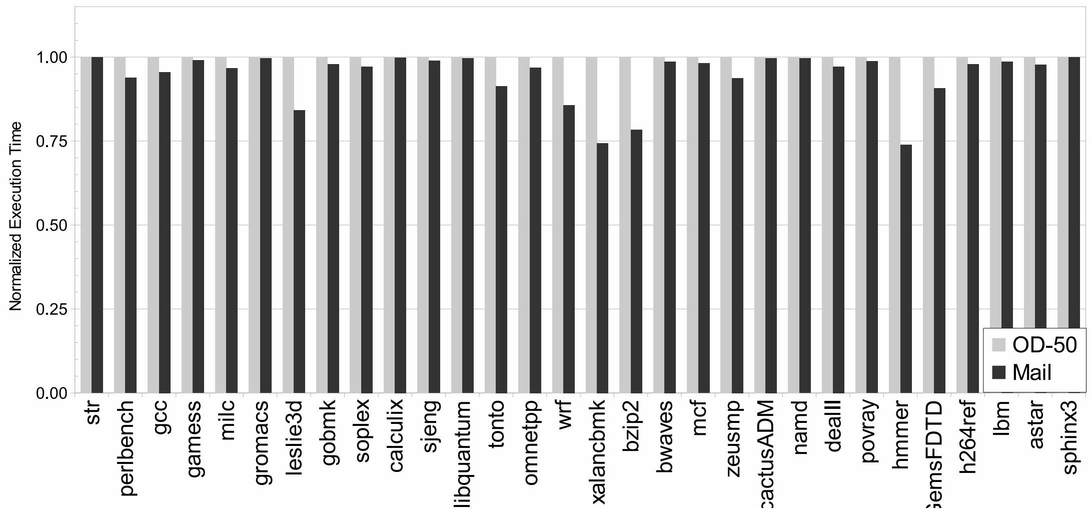
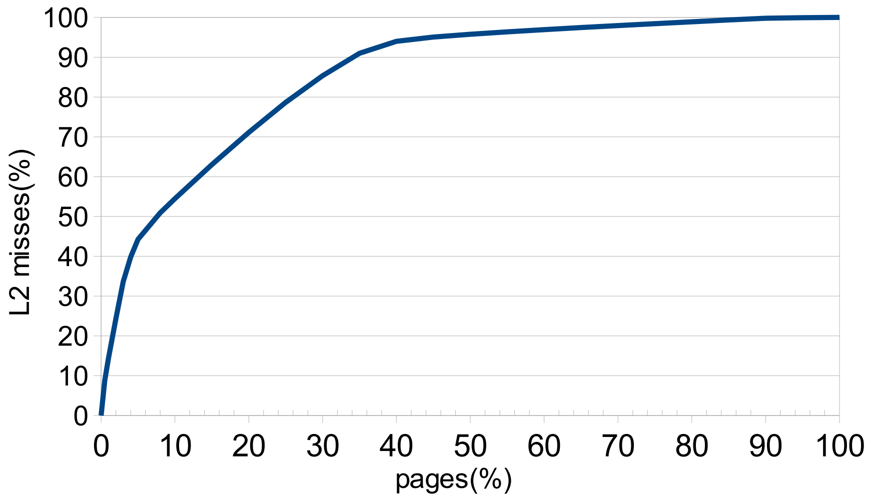


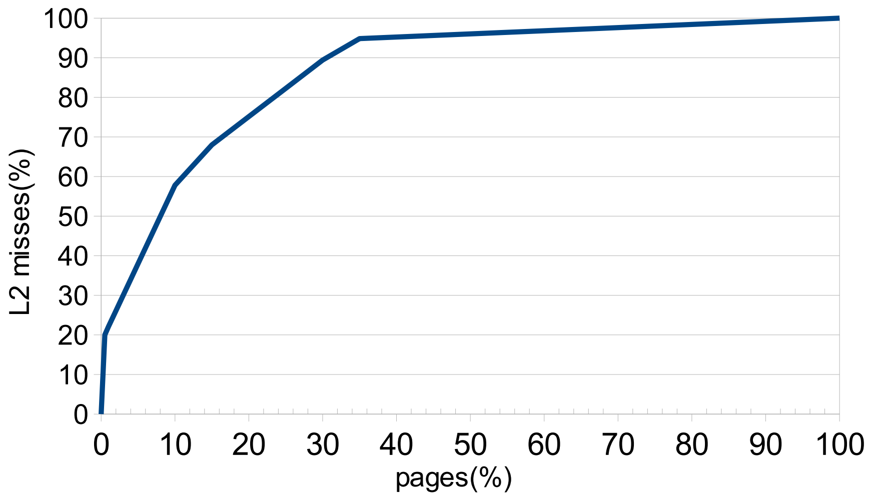
FIGURE 6.6: Mail versus on-demand.

To understand why some benchmarks are more sensitive to the improved page distribution applied by Mail, Figure 6.7 and Figure 6.8 plot, for the same four benchmarks analyzed in Section 6.4, the accumulated distribution of L2 misses (i.e., main memory accesses) ordered along the X axis from the most accessed to the least accessed page.

For example, Figure 6.8(a) shows that for *hmmer*, the 20% most accessed pages are responsible of around a 100% of its L2 misses. Thus, *hmmer* performance is greatly improved by the Mail scheme, which places the most accessed pages in the local memory region. On the other hand, although *wrf* also benefits from the Mail distribution, its performance improvement is smaller, since the L2 misses are not concentrated in a small set of pages, as shown in Figure 6.8(b). Nevertheless, its performance improvement with the Mail allocation policy is by 15%.

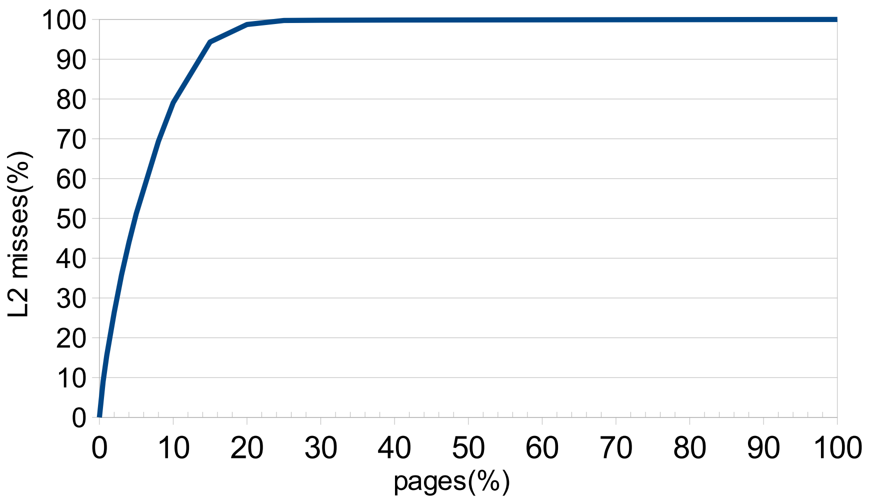


(a) astar

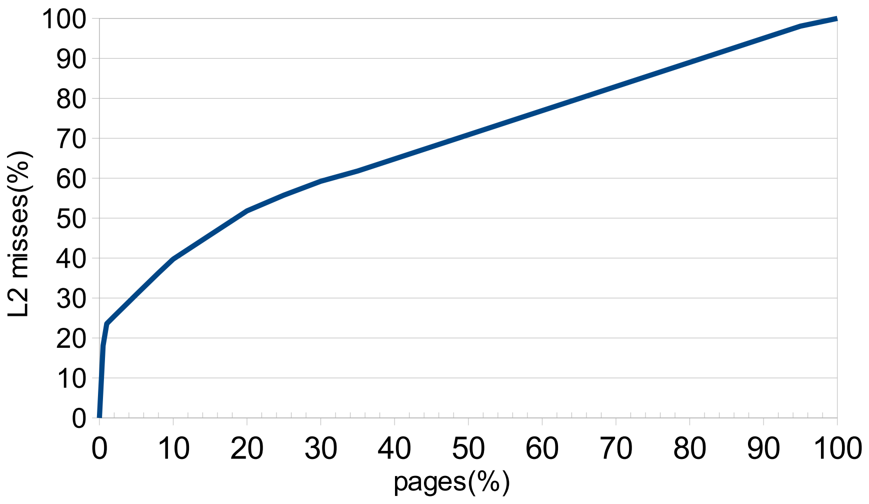


(b) bzip2

FIGURE 6.7: Accumulated percentage of main memory accesses. The best performing benchmarks under OD.



(a) hmmer



(b) wrf

Figure 6.8: Accumulated percentage of main memory accesses. The worst performing benchmarks under OD.

6.6 Summary

In this chapter, we have compared the performance of conventional hardware-based block interleaving between local and remote memory with the performance of OS-based page interleaving. We have found that only some applications are significantly affected by page-based interleaving. Thus, we have investigated the reasons that cause this impact on performance in order to design better OS-based memory allocation policies.

The results of this study show two interesting observations. First, we have found that most memory accesses are not evenly distributed among pages but a small subset of pages are responsible of most of the accesses. Second, the most accessed pages are usually requested during the first half of the execution time.

Based on this observations , we have proposed two memory allocation policies, namely on-demand (OD) and Most-accessed in-local (Mail). The first one is a simple strategy that works by placing new pages in local memory until this region is full. Consequently, it performs better when the most accessed pages are requested and allocated before than the least accessed ones, which is the common case, as proven in this work. Experimental results show that OD policy reaches around 25% performance improvement for some benchmarks with respect to a typical block interleaving memory system. However, under the OD policy, some benchmarks still allocate a large percentage of the least accessed pages to local memory. In contrast, the Mail allocation policy avoids this problem by using profile information to guide the allocation of new pages. Under this scheme, all the benchmarks show better performance than under block interleaving, and in some cases the performance offered by OD is improved as much as 25%.

Preliminar results of the work discussed in this chapter have been published in [48].

Chapter 7

Conclusions

Several memory scheduling policies have been proposed in this dissertation addressing cluster computers. These systems have been assumed to have three main memory regions whose access can incur widely different latencies. The aim of the proposed schedulers is to improve the overall performance in this scenario. In this chapter, the main contributions on each of these memory scheduling schemes are summarized, followed by a discussion about future working directions and an enumeration of the scientific publications related to this thesis.

7.1 Contributions

The devised memory schedulers have been presented in chapters four to six. The baseline system consists of main memory located in three main regions: local to node (L), local to board (Lb), and remote (R). In addition, it is assumed that a fast interconnection mechanism allows the access to remote memory.

In Chapter 4, an ideal algorithm and a heuristic strategy to assign main memory from the different memory regions have been proposed. Conclusions in this chapter address three main issues: i) analysis of memory requirements for individual applications, ii) performance interferences between different co-runners, and iii) a heuristic whose performance is close to an optimal scheduler.

- Regarding the first issue, the study has shown that, i) the memory distribution between L and Lb can impact on performance when no R memory is allocated, ii) the previous distribution has a slight effect on the performance when R is allocated, and iii) performance degradation widely varies among applications when allocating R memory.
- With respect to the second issue, this study has shown that the total performance is benefited when as much remote memory as possible is assigned with the following pattern: first to the application that least degrades its performance, then to the second one, and so on. However, this assignment strategy can lead to unacceptable performance degradation for some applications. Hence, a quality of service parameter has been defined for each application.
- Finally, regarding the latter issue, two memory scheduling algorithms have been proposed. An ideal algorithm, namely SSP, which makes a search of the optimal memory distribution for all the applications, and a heuristic strategy guaranteeing QoS that approximates the best local and remote memory distribution among applications have been presented. Results have shown that the memory

distribution provided by the heuristic is close or the same as the optimal distribution found by SPP, whose computational cost is prohibitive for a high number of applications.

In Chapter 5, an analytical model which is used as a performance predictor was devised to estimate the execution time of a given application for a given memory distribution. We first carried out a study to determine the main variables (associated to simulation events) of the model, and classified them as memory-region dependent and independent. The model assumes that the number of cycles spent in each considered event is obtained from some hardware counters of the target machine. The model is used to provide the input to a memory scheduler, which using this input, is able to dynamically choose the optimal target memory distribution for each application running in the system in order to achieve the best overall performance. The validation study shows that the dynamic predictor is very accurate, since its deviation from the real results is always lower than 5% and close to 0% across the experiments.

Finally, in Chapter 6 we have studied the feasibility of scheduling at OS page granularity instead of accessing memory at cache block size granularity. We compared the performance of conventional hardware-based block interleaving (between local and remote memory) with the performance of OS-based page interleaving. We found that only some applications are significantly affected by page-based interleaving. Thus, we investigated the reasons that cause this impact on performance in order to design better OS-based memory allocation policies. Based on the results, we have proposed two memory allocation policies, namely on-demand (OD) and Most-accessed in-local (Mail). The first one is a simple strategy that works by placing new pages in local memory until this region is full. Thus, it performs better when the most accessed pages are requested and allocated before than the least accessed ones, which is the common case, as proven in this chapter. Experimental results showed that OD policy reaches around 25% performance improvement for some benchmarks with respect to a typical block interleaving memory system. However, under the OD policy, some benchmarks still allocate a large percentage of the least accessed pages to local memory. In contrast, the Mail allocation policy avoids this problem by using profile information to guide the allocation of new pages. Under this scheme, all the benchmarks show better performance than under block interleaving, and in some cases the performance offered by OD is improved as much as 25%.

7.2 Future Work

In the memory scheduling policies proposed in this thesis, the optimum memory distribution is assumed to be such that implies the lowest number of remote memory accesses, thus achieving better performance than a baseline memory distribution (i.e., a non-scheduled memory distribution).

As an extension of this idea, for future work on the topic of memory scheduling in cluster computers, we plan to design a dynamic memory scheduler aware of the most and the least accessed pages during the execution, using this information with the aim of suitably allocating them so that the main memory access time is minimized. That is, the most accessed pages will be allocated in the local memory region and the least accessed pages in the remote memory region. In this way, the scheduler can decrease the remote memory accesses as much as possible.

An important scheduling feature that must be addressed is the page migration problem, that is, pages in remote memory will be allowed to migrate to local memory, which can incur replacement of local pages. Page migrations involve some OS actions that may increase the overall execution time. Nevertheless, in most of cases, the time spent in page migration may be critical, especially for those workloads which change its memory access patterns through the execution. For instance, a memory page may be considered poorly accessed by the memory scheduler and hence allocated in the remote memory region in a given instant of the execution. After that, the accesses to that memory page (at a remote memory latency cost) may become more and more frequent leading to a bottleneck and severely damaging the overall system performance. Consequently, an alternative memory scheduling mechanism needs to be designed to keep track of the number of memory accesses to a given page in order to decide if either this page should stay in its current memory region or migrate to another region, from the performance point of view. More precisely, the baseline machine will be provided with page migration capabilities and several page memory placement and replacement techniques will be devised and evaluated.

The design of some of the aforementioned improvements opens new opportunities in the ground of efficient memory management in cluster computers. In fact, we are currently addressing these issues. Some memory data structures to keep dynamic track of the information about number of memory accesses, time since creation, last access, or current memory region for a given memory page have been already implemented on the baseline machine. In particular, it becomes an interesting research opportunity to design efficient memory page migration mechanisms. Regarding memory page placement, several techniques have been devised. The goal is to move a remote memory page to the local memory region when it has reached a given threshold estimated as the maximum number of accesses such that it is considered *too frequently accessed* to be remote. In some cases, the fact that a memory page reaches the threshold may be not enough to make the right placement decision, hence more specific requirements with higher level of difficulty (and thus more computational cost) should be taken into account. This remote-to-local or placement migration, in turn, may imply a local memory eviction (i.e., replacement) when there is no free space in the local region. Several techniques have been drafted to select the victim local page. A simple strategy will be applying the LRU (i.e., Least Recently Use) criterion. A more sophisticated technique may be to implement a *victim pages register* which efficiently combines LRU with additional information about the number of accesses for a given memory page, providing a limited set of memory pages among which a victim can be chosen at low computational cost. Finally, the combination of several techniques (e.g., Not Frequently Used plus Aging [49]) offers another alternative to study.

In this scenario, new challenges arise, as it is not enough to focus on decreasing the number of remote memory accesses but also to minimize the number of page migrations since, as stated above, placements and replacements are expensive from the execution time point of view. Hence, each of the alternatives must be evaluated in order to achieve a *number of remote accesses–number of migrations* tradeoff which leads to the optimal overall system performance.

In summary, scheduling of local and remote memory in cluster computes is still an interesting topic that can help to increase the overall performance in current systems, mainly by exploiting issues aimed at dynamizing the scheduling mechanism and adjusting the scheduling techniques to the specific memory access pattern exhibited by the applications.

7.3 Publications

The following list enumerates the papers related with this dissertation that have been published in specialized international conferences or journals.

- M. Serrano, J. Sahuquillo, S. Petit, H. Hassan, and J. Duato, "A Cost-Effective Heuristic to Schedule Local and Remote Memory in Cluster Computers", in *The Journal of Supercomputing*, volume 59, issue 3, pages 1533-1551, 2012.
- M. Serrano, J. Sahuquillo, H. Hassan, S. Petit, and J. Duato, "A Scheduling Heuristic to Handle Local and Remote Memory in Cluster Computers", in *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications* (HPCC), pages 35-42, Melbourne (Australia), September 2010.
- M. Serrano, J. Sahuquillo, H. Hassan, S. Petit, and J. Duato, "A Cluster Computer Performance Predictor for Memory Scheduling", in *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing* (ICA3PP), pages 353-362, Melbourne (Australia), October 2011.
- M. Serrano, S. Petit, J. Sahuquillo, R. Ubal, H. Hassan, and J. Duato, "Page-Based Memory Allocation Policies of Local and Remote Memory in Cluster Computers", in *Proceedings of the 18th International Conference on Parallel and Distributed Systems* (ICPADS), pages 612-619, Singapore (Singapore), December 2012.

In addition, other related papers have been published in domestic conferences:

- M. Serrano, J. Sahuquillo, H. Hassan, S. Petit, and J. Duato, "Una Heurística de Planificación de Memoria Local y Remota en Clústers de Computadores", in *Actas de las XXI Jornadas de Paralelismo* (JP), pages 357-364, València, Spain, September 2010.
- M. Serrano, J. Sahuquillo, H. Hassan, S. Petit, and J. Duato, "Predictor de Prestaciones para la Planificación de Memoria en Clústers de Computadores", in *Actas de las XXIII Jornadas de Paralelismo* (JP), pages 400-405, Elx, Spain, September 2012.

All the works listed above are exclusively related with this thesis, and none of them are or will be used as supporting material for other theses. The specific contributions of the Ph.D. candidate reside mostly in the implementation of the proposed techniques, the setup and execution of most simulation experiments, and the writing of the paper drafts describing the work. Along these processes, the coauthors have repeatedly provided useful hints and advices, which the Ph.D. candidate has then applied to make the work evolve into its final version. All the conference papers listed above were presented and defended by the Ph.D. candidate.

Finally, the acquired skills by the Ph.D. candidate during the development of this work, have been also applied at laboratory sessions in the Advanced Computer Architectures Course of the Computer Engineer Degree offered by the School of Computer Engineering at*Universidad Politécnica de Valencia* during the 2010-2011, 2011-2012 and 2012-2013 academic years as established in her *FPU* fellowship from the Spanish Ministry of Education (Training Program for University Teachers). Related with the theoretical and practical contents of this course, the following paper was published in collaboration with other members of the research group.

- C. Gómez, M. Serrano, M. E. Gómez, and J. Sahuquillo, "Una Nueva Metodología para el Estudio de Procesadores Realistas en las Titulaciones de Informática", in *Actas de las XXIII Jornadas de Paralelismo* (JP), pages 507-512, Elx, Spain, September 2012.

References

[1] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* 2006.

[2] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture - a Hardware / Software Approach.* 1999.

[3] Numascale's NumaConnect^{TM}technology [online]. Available from: `http://www.numachip.com`.

[4] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed Shared Memory: Concepts and Systems. *Parallel Distributed Technology: Systems Applications, IEEE*, 4(2):63–71, 1996.

[5] D. Geer. Industry Trends: Chip Makers Turn to Multicore Processors. *IEEE Computer*, 38(5):11–13, 2005.

[6] IBM System z [online]. Available from: `http://www.ibm.com/systems/z`.

[7] HP Integrity Servers [online]. Available from: `http://h20341.www2.hp.com`.

[8] H. Litz, H. Fröening, M. Nuessle, and U. Brüening. A HyperTransport Network Interface Controller for Ultra-low Latency Message Transfers. *HyperTransport Consortium White Paper*, 2007.

[9] M. Nussle, M. Scherer, and U. Bruning. A Resource Optimized Remote-Memory-Access Architecture for Low-latency Communication. In *Proceedings of the International Conference on Parallel Processing*, 2009.

[10] M. Blocksome, C. Archer, T. Inglett, P. McCarthy, M. Mundy, J. Ratterman, A. Sidelnik, B. Smith, G. Almási, J. Castanos, D. Lieber, J. Moreira, S. Krishnamoorthy, V. Tipparaju, and J. Nieplocha. Design and Implementation of a One-sided Communication Interface for the IBM eServer Blue Gene®supercomputer. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 2006.

[11] S. Kumar, G. Dózsa, G. Almasi, P. Heidelberger, D. Chen, M. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. E. Smith, and C. Archer. The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, 2008.

[12] V. Tipparaju, A. Kot, J. Nieplocha, M.T. Bruggencate, and N. Chrisochoides. Evaluation of Remote Memory Access Communication on the Cray XT3. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, 2007.

[13] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. S. Szalay, D. J. DeWitt, and G. Heber. Scientific Data Management in the Coming Decade. *ACM SIGMOD Record*, 34(4):34–41, 2005.

[14] The Official Gaussian Website [online]. Available from: `http://www.gaussian.com`.

[15] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *Proceedings of the IEEE 18th International Symposium on High-Performance Computer Architecture*, 2012.

[16] H. Midorikawa, M. Kurokawa, R. Himeno, and M. Sato. DLM: A Distributed Large Memory System using remote memory swapping over cluster nodes. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2008.

[17] H. Midorikawa and J. Uchiyama. Automatic Adaptive Page-Size Control for Remote Memory Paging. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2012.

[18] S. Liang, R. Noronha, and D. K. Panda. Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2005.

[19] P. Werstein, X. Jia, and Z. Huang. A Remote Memory Swapping System for Cluster Computers. In *Proceedings of the 8th International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2007.

[20] J. Oleszkiewicz, L. Xiao, and Y. Liu. Parallel Network RAM: Effectively Utilizing Global Cluster Memory for Large Data-Intensive Parallel Programs. In *Proceedings of the International Conference on Parallel Processing*, 2004.

[21] K. Jeon, H. Han, S. Kim, H. Eom, H.Y. Yeom, and Y. Lee. Large Graph Processing Based on Remote Memory System. In *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications*, 2010.

[22] KM. rishnan, R.R. Lewis, and A. Vishnu. Scaling Linear Algebra Kernels Using Remote Memory Access. In *Proceedings of the 39th International Conference on Parallel Processing Workshops*, 2010.

[23] M. Oguchi and M. Kitsuregawa. Dynamic Remote Memory Acquisition for Parallel Data Mining on ATM-connected PC Cluster. In *Proceedings of the 13th International Conference on Supercomputing*, 1999.

[24] M. Oguchi and M. Kitsuregawa. Using Available Remote Memory Dynamically for Parallel Data Mining Application on ATM-Connected PC Cluster. In *Proceedings of the 14th International Parallel & Distributed Processing Symposium*, 2000.

[25] P. Lu, Y. Che, and Z. Wang. A Framework for Effective Memory Optimization of High Performance Computing Applications. In *Proceedings of the 11th IEEE International Conference on High Performance Computing and Communications*, 2009.

[26] Y. Xie and G. H. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. *2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*

in conjunction with the 35th International Symposium on Computer Architecture, 2008.

[27] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. Cache Contention and Application Performance Prediction for Multi-core Systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2010.

[28] J.K. Rai, A. Negi, R. Wankar, and K.D. Nayak. Performance Prediction on Multicore Processors. In *Proceedings of the International Conference on Computational Intelligence and Communication Networks*, 2010.

[29] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[30] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[31] C. D. Antonopoulos, D.S. Nikolopoulos, and T.S. Papatheodorou. Scheduling Algorithms with Bus Bandwidth Considerations for SMPs. In *Proceedings of the International Conference on Parallel Processing*, 2003.

[32] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Realistic Workload Scheduling Policies for Taming the Memory Bandwidth Bottleneck of SMPs. In *Proceedings of the 11th International Conference on High Performance Computing*. 2004.

[33] E. Koukis and N. Koziris. Memory and Network Bandwidth Aware Scheduling of Multiprogrammed Workloads on Clusters of SMPs. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, 2006.

[34] D. S. Nikolopoulos. Quantifying and Resolving Remote Memory Access Contention on Hardware DSM Multiprocessors. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002.

[35] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron Processor for Multiprocessor Servers. *IEEE Micro*, 23(2):66–76, 2003.

[36] J. Duato, F. Silla, and S. Yalamanchili. Extending HyperTransport Protocol for Improved Scalability. In *Proceedings of the 1st International Workshop on HyperTransport Research and Applications*, 2009.

[37] SPEC CPU2006 Benchmark Descriptions. *SACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[38] C. D. Spradling. SPEC CPU2006 Benchmark Tools. *SIGARCH Computer Architecture News*, 35, 2007.

[39] S. Cameron Woo, M. Ohara, E. Torrie, J. Pal Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd annual International Symposium on Computer architecture*, 1995.

[40] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. Available from: `http://www.cs.virginia.edu/stream/`.

[41] D. H. Bailey. FFTs in External or Hierarchical Memory. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 1989.

[42] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. Greg Plaxton, S. J. Smith, and M. Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. *Commun. ACM*, 39(12es):273–297, 1996.

[43] R. Ubal, J. Sahuquillo, S. Petit, and P. López. Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. In *Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing*, 2007.

[44] Multi2Sim: A CPU-GPU Simulator for Heterogeneous Computing [online]. Available from: `http://www.multi2sim.org/`.

[45] A. Levitin. *Introduction to The Design and Analysis of Algorithms.* Addison Wesley, 2003.

[46] M. Serrano, J. Sahuquillo, S. Petit, H. Hassan, and J. Duato. A Cost-effective Heuristic to Schedule Local and Remote Memory in Cluster Computers. *The Journal of Supercomputing*, 59(3):1533–1551, 2012.

[47] M. Serrano, J. Sahuquillo, H. Hassan, S. Petit, and J. Duato. A Cluster Computer Performance Predictor for Memory Scheduling. In *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing, Part II*, pages 353–362, 2011.

[48] M. Serrano, S. Petit, J. Sahuquillo, R. Ubal, H. Hassan, and J. Duato. Page-Based Memory Allocation Policies of Local and Remote Memory in Cluster Computers. In *Proceedings of the 18th International Conference on Parallel and Distributed Systems*, pages 612–619, 2012.

[49] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems - Design and Implementation (3. ed.).* Pearson Education, 2006.