Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Master's Thesis Nr. 10

Systems Group, Department of Computer Science, ETH Zurich

Performance isolation on multicore hardware

by

Kaveh Razavi

Supervised by

Akhilesh Singhania and Timothy Roscoe

November 2010 - May 2011

inf | Informatik
Computer Science

# *Abstract*

Modern multicore systems differ from commodity systems in the way system resources are shared. Inter-processor main memory bandwidth, different levels of caches and the system interconnect among other resources are now shared across different cores in the system. Allocation of these shared resources is not always in the direct control of the operating system. As a result, applications are allocated certain resources by the operating system, but they implicitly share other resources with other applications in the system. The lack of operating system control over this implicit resource sharing results in undesired performance degradation of performance critical applications. In this thesis, we investigate the feasibility of running applications under *performance isolation* in modern multicore systems. We show one of the possible ways for providing performance isolation in such systems by 1) controlled placement of applications on cores and memory locations and 2) avoiding contention on the memory subsystem. We then design a resource management subsystem with mechanisms for performance isolation and performance-aware resource allocation on top of a multicore operating system. At the end, we show the benefits of having such a subsystem in multicore operating systems.

# *Acknowledgements*

I would like to thank my advisor Prof. Timothy Roscoe who made it possible for me to come to ETH Zürich. I always had his support from the beginning of my master studies and I certainly learned a great deal from him.

I would like to thank my mentor Akhilesh Singhania for helping me understand the Barrelfish's internals. Without his dedication and continuous feedback, this thesis was not possible.

I would also like to mention Adrian Schüpbach and Simon Peter for answering my many questions on Barrelfish during the course of this thesis and my lab project.

Last but not least, I would like to thank Hanieh, for her support during the course of my stay in Switzerland and also proofreading this thesis.

# Contents


**Abstract** **i**

**Acknowledgements** **ii**

**List of Figures** **vi**

**List of Tables** **viii**

**1 Introduction** **1**

**2 Background work** **3**

2.1 Literature review . . . . . . . . . . . . 3

2.1.1 Operating systems that aim at performance isolation . . . . . . . . 4

2.1.2 Providing mechanisms for performance isolation . . . . . . . . . . 4

2.1.3 Reducing contention over shared resources . . . . . . . . . . . . . 5

2.1.3.1 Reducing the contention over shared cache . . . . . . . . 6

2.1.3.2 Multicore scheduling to reduce contention over memory subsystem . . . . . . . . . . . . 7

2.1.4 Multicore benchmarks . . . . . . . . . . . . 8

2.2 Barrelfish . . . . . . . . . . . . 9

2.2.1 System Knowledge Base . . . . . . . . . . . . 10

2.3 Linux and performance isolation . . . . . . . . . . . . 10

2.3.1 System topology on sysfs . . . . . . . . . . . . 10

2.3.2 NUMA memory allocation through libnuma . . . . . . . . . . . . 10

2.3.3 User mode process placement and priority enforcement . . . . . . . 11

2.4 Summary . . . . . . . . . . . . 12

**3 A review of current multicore processors** **13**

3.1 Current multicore architectures . . . . . . . . . . . . 14

3.1.1 AMD Magny-Cours . . . . . . . . . . . . 15

3.1.2 Intel Nehalem-EX . . . . . . . . . . . . 17

3.2 Cache coherency and cache directory . . . . . . . . . . . . 18

3.3 STREAM on modern multicore systems . . . . . . . . . . . . 19

3.4 Summary . . . . . . . . . . . . 20

**4 Resource sharing and performance in multicore systems** **22**
4.1 A synthetic microbenchmark . . . . . 22
4.1.1 Processing the results . . . . . 25
4.1.2 Experiment environment . . . . . 25
4.1.3 Reading experiment setup diagrams . . . . . 26
4.2 Cores sharing a memory controller . . . . . 26
4.3 Effects of the shared interconnect . . . . . 28
4.3.1 Performance degradation caused by coherency messages . . . . . 28
4.3.2 Performance degradation caused by routing data . . . . . 33
4.3.3 Performance degradation caused by sharing a link . . . . . 35
4.4 Hardware threads sharing a core . . . . . 37
4.5 Summary . . . . . 39

**5 Performance isolation support in a multicore operating system** **40**
5.1 Enforcing performance isolation . . . . . 40
5.2 Enforcing memory isolation . . . . . 42
5.2.1 Memory bandwidth isolation and process migration . . . . . 43
5.3 Performance degradation caused by sharing L3 . . . . . 44
5.4 Summary . . . . . 48

**6 A resource management subsystem for Barrelfish** **49**
6.1 Representation of memory subsystem in Barrelfish . . . . . 49
6.2 A resource management subsystem for Barrelfish . . . . . 51
6.2.1 Microbenchmarks to detect memory performance properties . . . . . 51
6.2.2 Monitoring memory bandwidth consumption . . . . . 52
6.2.3 System topology using the information stored in SKB . . . . . 53
6.2.4 An algorithm to create the interconnect topology . . . . . 54
6.2.5 Resource management using Barrelfish capability system . . . . . 55
6.2.6 An API for multicore resource management . . . . . 56
6.2.7 A library for interacting with RCM . . . . . 57
6.2.8 Limitations of RCM . . . . . 58
6.3 Summary . . . . . 59

**7 Evaluation** **61**
7.1 PARSEC benchmark suite . . . . . 61
7.2 Benchmarking performance properties . . . . . 63
7.3 Benchmarking isolation properties . . . . . 65
7.4 Summary . . . . . 69

**8 Conclusion** **70**
8.1 Future work . . . . . 71

**A Terminology** **73**

**B RCM's API** **75**

**C Using RCM within fish** **79**

**Bibliography** **80**

# List of Figures


2.1 The multikernel model. . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

3.1 One of Magny-Cours' dies. . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.2 Magny-Cours processor architecture. . . . . . . . . . . . . . . . . . . . . . 14
3.3 Interconnect topology of a machine with four Magny-Cours processors. . . 15
3.4 Magny-Cours crossbar switch architecture. . . . . . . . . . . . . . . . . . . 16
3.5 Intel Nehalem-EX processor die. . . . . . . . . . . . . . . . . . . . . . . . 17
3.6 Interconnect topology of a machine with eight Nehalem-EX processors. . . 18

4.1 The experiment setup for the memory controller experiment. . . . . . . . 26
4.2 Memory controller experiment results. . . . . . . . . . . . . . . . . . . . . 27
4.3 The experiment setup when routing coherency messages. . . . . . . . . . . 28
4.4 The experiment setup when not routing coherency messages. . . . . . . . 29
4.5 Performance degradation caused by coherency traffic. . . . . . . . . . . . . 30
4.6 Performance degradation caused by coherency traffic. . . . . . . . . . . . . 30
4.7 The experiment setup for coherency traffic of all cores. . . . . . . . . . . . 32
4.8 Performance degradation caused by coherency traffic of all cores. . . . . . 32
4.9 Experiment setup when routing data of load threads. . . . . . . . . . . . . 33
4.10 Experiment setup when data is routed through load threads' node. . . . . 34
4.11 Performance degradation of different data routing scenarios. . . . . . . . . 35
4.12 Experiment setup of sharing interconnect link bandwidth. . . . . . . . . . 36
4.13 Sharing interconnect link bandwidth experiment result . . . . . . . . . . . 36
4.14 The SMT experiments' design sketches. . . . . . . . . . . . . . . . . . . . 37
4.15 Performance comparison of various SMT setups. . . . . . . . . . . . . . . 38

5.1 Experiment setup for shared cache performance degradation. (cache and DRAM) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
5.2 Experiment setup for shared cache performance degradation. (cache) . . . 46
5.3 L3 cache miss rate comparison of local and remote stramclusters . . . . . 46
5.4 Runtime degradation of the benchmark in two scenarios . . . . . . . . . . 47
5.5 Decomposition of runtime degradation by memory subsystems . . . . . . 47

6.1 Topology representation in RCM . . . . . . . . . . . . . . . . . . . . . . . 53
6.2 Execution steps with RCM . . . . . . . . . . . . . . . . . . . . . . . . . . 56

7.1 The memory bandwidth usage of PARSEC benchmarks . . . . . . . . . . 62
7.2 The L3 cache miss rate of PARSEC benchmarks . . . . . . . . . . . . . . 63
7.3 Runtime comparison of the PARSEC benchmarks . . . . . . . . . . . . . . 64
7.4 Performance isolation experiment with PARSEC . . . . . . . . . . . . . . 67

7.5 Performance isolation experiment with multiple isolated instances . . . . . 69

# List of Tables

3.1 STREAM benchmark rating . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.2 STREAM benchmark rating without cache directory . . . . . . . . . . . . . 20

# Listings

4.1 Core of the measuring thread . . . 23
5.1 Estimation of availability of memory bandwidth for a new core . . . 44
6.1 Interconnect path discovery algorithm . . . 54
B.1 RCM Interface file . . . 75
C.1 Invocation of rcm in Fish . . . 79

*To my family*

# Chapter 1

# Introduction

One of the primary tasks of an operating system is resource allocation; that is, allocating the resources of a system (such as CPU time, memory, disk storage or bandwidth, etc.) to competing applications [1]. In modern multicore systems, some of these resources are implicitly shared and their allocation is not directly under the control of the operating system. For example, cores may share some levels of cache and in most of the current systems such as the ones with AMD and Intel processors, it is not possible for the operating system to allocate these caches to cores. Another example is when cores access different locations in memory through the system interconnect; allocation of different units of the interconnect to cores is also not explicitly under the control of the operating system. As a result of this complex hierarchy of resources which may or may not be shared, it is not clear to what extent the operating system can provide true resource allocation. The lack of operating systems' control over shared resources can affect the performance of an application that is explicitly allocated some resources and implicitly shares other resources with other applications.

In this thesis, we investigate to what extent performance isolation is possible in modern multicore systems and possible ways for a multicore operating system to provide performance isolation. *Performance isolation* is providing guaranteed resource allocation to an application so that it is not affected (or minimally affected) when it shares the internal system resources with other applications. Performance isolation is possible if 1) true resource allocation can be provided and 2) the applications are given unlimited access to *all* of their isolated resources during the course of their execution.

The contributions of this thesis are the following items:

[1]In this thesis, by application we mean a set of processes executing together to perform a certain task. Thus, a process is part of an application.

- We identify different levels of resource sharing by looking at two modern multicore processors. Then, by running different microbenchmarks, we look at the effects of this sharing on performance.
- Based on these results, we come up with concrete conditions under which performance isolation can be provided in a multicore system.
- We discuss possible designs of a resource management subsystem which provides performance isolation support for multicore operating systems and we describe an implementation of such a subsystem for a multocore operating system.
- We investigate the benefits of our implemented resource management subsystem by using a mix of macrobenchmarks.

In chapter 2, we look at background work on performance isolation. We also explore Barrelfish [1], the operating system which this work implements upon and performance isolation support in Linux. In chapter 3, we study two modern multicore processors and then, we identify the resources that are being shared, explicitly or implicitly. Chapter 4 then carefully investigates the effect of this sharing on performance and its indications for performance isolation and we come up with concrete conditions required for performance isolation. In chapter 5, we discuss how a multicore operating system should provide support for performance isolation. Chapter 6 describes the design and implementation of a resource management subsystem for Barrelfish. Chapter 7 evaluates the implemented functionality for performance isolation in Barrelfish and chapter 8 concludes.

# Chapter 2

# Background work

Performance isolation has been the topic of research since mid 90s and nowadays, it is even more in the center of attention because of increased resource sharing as a result of increased parallelism offered by modern multicore hardware. The research topics range from providing some form of quality of service (QoS) for network traffic to reducing crosstalk on processor caches. In this section, we briefly go through the background work on performance isolation and then we describe Barrelfish, a multicore operating system and its components related to this thesis in more detail. We also take a closer look at Linux, a popular open source operating system in research community and its mechanisms for providing performance isolation.

## 2.1 Literature review

We distinguish different research work related to performance isolation in four categories. The work in this thesis is built upon the ideas presented here and this rough categorization helps putting it in context:

1. Operating systems that aim at performance isolation
2. Operating systems that provide mechanisms for performance isolation
3. Reducing contention over shared resources in the system
4. Benchmarks which can highlight performance isolation properties of multicore operating systems

### 2.1.1 Operating systems that aim at performance isolation

We describe the operating systems with performance isolation as a design principle.

Some of the operating systems with real-time support are examples of such designs. The main requirement of real-time applications is to meet some form of deadline. To do so, they need to have uninterrupted access to their allocated resources. Therefore, they need to have some degree of performance isolation. Providing resource QoS guarantees for soft real-time applications like continuous media (audio/video streams) applications is one of the early attempts to provide performance isolation. Rialto OS [2] tries to provide modular and distributed real-time resource management to allow for multiple real-time applications to co-exist in system with shared resources. This is made possible by 1) adding appropriate real-time programming abstractions (like resource type and resource amount), which allow for applications to reason about their resource requirements and 2) a system resource planner to control the shared resources between competing applications.

Applications usually know more about their resource requirements than the operating system. As a result, application-level resource management is beneficial for applications since they can use specific policies for resource management according to their needs. It is also beneficial for the operating system since resource management policies are in the applications themselves instead of one centralized operating system service. Exokernel [6] is an operating system architecture for application-level resource management. In Exokernel, an application is given unlimited access to the system resources with minimal kernel intervention and there is no policy in the kernel regarding the way an application uses system resources. This interesting design allows for implementing performance isolation policies in the application rather than worrying about them in the kernel.

Tessellation [10] is a multicore operating system which argues for space-time partitioning of resources in the system. In Tessellation, each application is given unlimited access to space partitions which are isolated set of hardware resources. Each application within its partition can enforce resource isolation policies. What is not clear yet is how the operating system should enforce isolation in space partition granularity since as we will discuss in chapter 3, in a multicore processor, some of the resources in the system can inherently be shared across partitions.

### 2.1.2 Providing mechanisms for performance isolation

A different approach is by using mechanisms the operating system provides for performance isolation. Applications can use these mechanisms to control the resource usage

of their different building blocks.

Banga et al. [11] argue that mis-accounting of resource consumption leads to incorrect scheduling based on their observation of different webserver architectures. To solve this problem, they propose a novel approach for resource management on a commodity UNIX operating system which distinguishes between process abstraction and resource usage. Resource containers are proposed as a new abstraction for resource management with which they enforce per process resource allocation policies. These mechanisms can enable applications to implement policies for providing different resource QoS guarantees among other desirable properties.

Our resource management subsystem described in chapter 6, makes resource allocation explicit in a similar way and provides similar benefits as resource containers. For example, an application can request different resources with different performance isolation guarantees (like isolated cores or isolated caches) and use them at the time of its choosing to provide different QoS guarantees or minimize contention over the resources that it holds.

There are other mechanisms for providing performance isolation by controlling the contention over the shared resources discussed in the next section.

### 2.1.3 Reducing contention over shared resources

Reducing the contention over shared resources is one of the requirements for providing performance isolation. This can either be automatically supported by the operating system, or it can be a mechanism that applications can use. We discuss these cases here.

Bellosa [3] observed that the quality of video conferencing which is measured in frames per second, occasionally degrades as other processes in the system use the memory bandwidth. He explored possible ways for controlling process memory access rate in UNIX System V. He argued for preemption based on memory bandwidth usage in shared memory bandwidth multiprocessors as a mechanism to provide memory bandwidth QoS for soft real-time streaming applications like video conferencing. His method was effective to provide memory bandwidth QoS for streaming media applications.

Marchand et al. [4] designed a method to provide memory QoS guarantees for real-time applications. They implemented a dynamic memory allocator which interacts with the memory controller. If the memory controller cannot meet the deadline of real-time applications, then some of the memory allocations will have to wait and will go to a failed request queue. This decision is based on a model which is proven to perform well in overloaded systems [5]. The scheduling algorithm uses the failed queue statistics to

make scheduling decisions. This method was developed for uniprocessor systems and memory controllers in the current multicore systems which we explored, do not provide such facilities.

Nemesis OS [7] is an exokernel designed with multimedia applications in mind. Operating systems usually provide low level memory management (paging) in one of their components to fulfill applications requests for virtual to physical mappings. However, Nemesis takes a different approach to this problem. By handling page faults at application level [8], Nemesis OS can control the QoS crosstalk over the operating system pager. Barham et al. [9] take the idea further to virtualization technology by suggesting that each guest operating system should perform its own paging using its own guaranteed memory reservation and disk allocation. These are other examples of benefits of application-level resource management.

Our implementation of the resource management subsystem described in chapter 6, allows for application-level management of the system resources, but it also makes sure that performance isolation guarantees are not violated. The distributed shared nothing approach of Barrelfish, along with its similarities with the Exokernel architecture, voids centralized contention of low level system resource management like paging. We will talk more about Barrelfish in section 2.2. Moreover, in section 5.2.1, we will describe our method of controlling the shared memory bandwidth usage.

#### 2.1.3.1 Reducing the contention over shared cache

Different levels of cache hide the latency of main memory. In many modern multicore systems, some of these caches (usually the last level cache) are shared between some cores. As a result, the applications running on different cores with a shared cache can have performance impact on each other. Polluting the shared cache of other cores (or other processes) is a common phenomenon observed when the cache is shared and it results in considerable performance degradation. The research efforts discussed here try to address this problem.

On the processor on-chip cache, cache partitioning [12] was proposed to provide isolation on the shared cache between competing processes in the system. However, this method needs to be implemented in hardware and common multicore processors do not provide such facilities. With increased sharing of cache between cores in modern multicore processors, providing such facilities is becoming even more important.

Ahsan et al. [13] pointed out that in modern multicore processors, cores share off-chip bandwidth and this bandwidth is becoming a more severe performance bottleneck when

many cores share it. Thus, they argue for a bandwidth-friendly replacement policy for shared caches instead of traditional LRU replacement policy.

Lee et al. [14] proposed cache coloring for minimizing cache conflicts in multicore processors for databases. This method tries to provide a similar facility as that of cache partitioning, but in software.

Non-Uniform Cache Architecture (NUCA) proposed by J. Lira et al. [15] is another method proposed and implemented in hardware to isolate the effect of delay when fetching memory from remote processors. NUCA works by partitioning the cache into banks with different speeds and using the faster ones for local memory.

As discussed in this section and in other relevant publications (e.g. [16–18]), shared caches are the source of some performance problems and there have been many attempts to reduce cache contention, both in software and hardware. What is missing in commonly used multicore hardware, is support for controlling the shared cache in software.

We study the degradation caused by shared cache in section 5.3 and in section 6.2.6 we provide a primitive to isolate the shared cache in our proposed resource management subsystem by not allocating any other process on the cores which share cache. While this primitive can result in serious under-utilization of the processor, it provides a mechanism for applications that suffer the most by cache pollution to run in an isolated cache. In section 7.3, we evaluate the benefits and shortcomings of isolating the shared cache.

#### 2.1.3.2 Multicore scheduling to reduce contention over memory subsystem

As discussed in chapter 4, current multicore systems implicitly share internal resources to access memory. This can result in contention over these resources. One important question is how to reduce this contention as much as possible. Here, we discuss proposed answers to this question, which are specific to modern multicore systems.

J. Lira et al. [19] studied the effect of memory sharing in multicore processors. They showed that in a system which needs Front System Bus (FSB) to access an off-chip memory controller, performance degradation should be expected when accessing memory from different cores. In the best case scenario, when the FSB, bus controller and the cache are not shared between two cores, the STREAM benchmark [20] still shows the degradation of 10%. They do not discuss the reasons for this degradation but we speculate this is mostly because of coherency messages. When the FSB is shared, the performance drops by around 50%. Based on these results, to improve performance, they have created a user-mode scheduler which dynamically places processes on different cores to reduce the memory path sharing of memory intensive processes. In most

modern multicore processors, the memory controller is usually on-chip and the FSB is no longer the bottleneck.

S. Blagodurov et al. [21] investigated the benefits of contention-aware scheduling for multicore systems. They showed that the performance degradation caused by pollution over the shared cache in multicore systems is only a fraction of the total performance degradation. Most of the degradation is caused by contention over the memory controller, the prefetcher, the interconnect and the FSB if present. They suggested that reducing the cache miss rate of last level shared cache would reduce most of the contention over the memory subsystem. Based on their findings, they designed different contention-aware scheduling algorithms to improve performance or to reduce the power usage of the system. Their main algorithm tries to equalize the cache miss rate of the last level cache of all processors in the system by changing the location of processes on different processors. However, their algorithm does not guarantee any form of performance isolation, just the best possible way to reduce contention for improved performance.

It is clear by these recent research work that lack of control over shared memory bandwidth can result in an unfair performance degradation of some memory sensitive applications. Since the last level cache miss rate correlates with the amount of shared memory bandwidth, both of these proposed solutions use scheduling based on the shared bandwidth consumption to minimize the contention over the memory subsystem.

We take away some important novelties from this part as well. We monitor the shared memory bandwidth in order to make sure that performance isolation guarantees are not violated (in section 5.2.1). To do so, we provide (in section 6.2.1) small microbenchmarks which measure the actual shared bandwidth provided to each core by each memory controller in the system.

### 2.1.4 Multicore benchmarks

With the invention of multicore processors, multicore operating systems started appearing as means for researchers to implement their design ideas on these new systems. Benchmarking has long been a method of showing different characteristics of special or general-purpose operating systems. It is a question that whether current benchmarks are adequate enough to show different properties of multicore operating systems.

Kuz et al. [23] argue that the current benchmarks that are used by the multicore operating system research community do not really highlight the interesting properties of such operating systems. They suggest that a multicore benchmark should consist of a mix of workloads, which all together exercise all resources of the system and thus,

can highlight important properties of a multicore operating system like performance isolation.

D. Hackenberg et al. [22] suggest that simple benchmarks like STREAM can no longer be used to compare new complex multicore processor architectures and therefore, they suggest a new benchmark to capture performance properties of such processors.

In section 4.1, we will describe a microbenchmark to show different performance properties of multicore systems. In chapter 7, we come up with a method of measuring the effectiveness of our performance isolation support using the ideas presented in these publications.

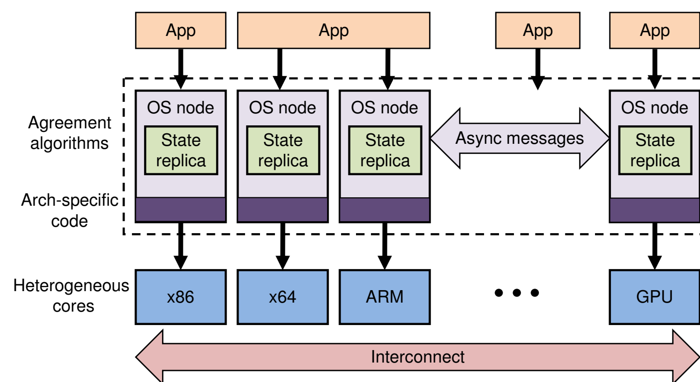


FIGURE 2.1: The multikernel model.

## 2.2 Barrelfish

Barrelfish is a multicore OS which is currently being developed as a joint project in ETH Zurich and Microsoft Research. Barrelfish is built upon the multikernel model [1]. The multikernel model takes a shared nothing approach in structuring the operating system. Each operating system node runs on an execution unit and communicates with other operating system nodes via asynchronous messages to maintain the replicated state of the system. Figure 2.1 shows the multikernel model in greater detail.

Barrelfish's architecture is interesting for studying performance isolation because most of the operating system's tasks are distributed between all the cores in the system (i.e. each core runs a different kernel). Barrelfish also tries to avoid centralized services that are prominent sources of contentions. Moreover, Barrelfish follows similar ideas of Exokernel architecture which minimize resource management policies in the kernel.

### 2.2.1 System Knowledge Base

System Knowledge Base (SKB) is a service within Barrelfish which provides a rich representation of the diverse and complex set of hardware[24]. To do so, it uses a prolog engine with a constraint solver [25] and stores the *facts* given to it in a database. This provides a unified database for storing information that the operating system, system servers and user applications can use to reason about the hardware. The SKB service also provides some logic to simplify querying common information (e.g. available memory addresses ranges). The facts are stored using different *schemas* for different types of information.

We will later use SKB to store some facts about the system which are important for providing performance isolation and better resource allocation according to application needs.

## 2.3 Linux and performance isolation

Linux is a popular operating system for researchers to present a work built on top of it or compare their developed prototypes with it. It is thus important to take a look at Linux and its support for performance isolation.

### 2.3.1 System topology on sysfs

Linux provides the system topology to user-mode applications through sysfs [26]. The NUMA information as well as shared cache information can be easily accessed through it. The kernel backend uses CPUID [28, 29] and ACPI tables [30] plus some hardcoded architectural facts to provide this information. For example, `cache_shared_cpu_map_setup` function in Linux kernel 2.6.37 always assumes that L3 caches are shared between all cores of the same node. If one of such facts does not hold in the diverse set of available hardware, it can lead to invalid entries in sysfs resulting in sub-optimal performance when an application uses this information.

Interconnect topology information is also missing in sysfs, which makes it impossible for applications to do interconnect-aware memory allocation.

### 2.3.2 NUMA memory allocation through libnuma

As we will discuss in the following chapter, both AMD and Intel, the giants of processor architectures, have moved the memory controller to the processor chip. This is to void

the FSB which rapidly becomes the bottleneck as the number of cores in the system increases. One direct result of this, is different memory latency and bandwidth when accessing memory through different memory controllers existing on different processors. This Non-Uniform Memory Access (NUMA) calls for careful memory allocation for different processes running on different processors for the sake of increased performance. Accessing local memory usually has lower latency and higher bandwidth. Accessing remote memory on the other hand, usually has higher latency, lower bandwidth and will utilize the shared interconnect.

Linux provides NUMA memory allocation support through a library called *libnuma* [31]. libnuma is a wrapper for a number of system calls to control the location of memory allocation for different processes. Linux kernel itself also supports NUMA for its own memory allocation to increase the performance of the kernel itself.

libnuma provides a number of functions in the form of `numa_alloc_XXX`, where XXX defines where the memory should be allocated from. For example, `numa_alloc_local` allocates memory on the NUMA node where the process has made the call and `numa_alloc_onnode` allocates memory on a specified NUMA node.

### 2.3.3 User mode process placement and priority enforcement

User-mode applications in Linux can use either *taskset* as an external application to control where they want to be placed, or they can internally use the scheduling routine *sched_setaffinity* to define it [33]. To use taskset, a process needs to provide a mask of the processors[1] where it is interested to execute on or alternatively, it can provide a list of the processors. For example, `taskset -c 0,1,2 program` restricts `program` to processors IDs zero, one and two.

It is also possible to change the scheduling priority using a functionality provided by the *nice* application. `nice -n value program` will change the niceness of `program` by `value`, which can either be positive or negative. Having a small "niceness" makes sure that the process is running with high priority, which results in minimized noise caused by other processes running in the system. Internally applications can use either the *nice* or *setpriority* (for a more fine grained scheduling requests) functions to do so [33].

[1] Linux treats each core as a different processor in this manner.

## 2.4 Summary

In this chapter, we looked at the background work in performance isolation. We describe different categories of research which are related to performance isolation:

- Operating systems that treat performance isolation as a requirement
- Operating systems that provide performance isolation mechanism for applications
- Reducing contention over shared resources in the system
- Benchmarks which can highlight performance isolation properties of multicore operating systems

We then looked at Barrelfish, the multicore operating system which we work with in this thesis. At the end, we looked at the mechanisms that Linux provides for performance isolation.

# Chapter 3

# A review of current multicore processors

Depending on the purpose of an application in the system, it uses different resources provided to it by the system. The sharing of resources like CPU, memory, disk, network and other devices is obvious. On systems with such resources, the operating system time multiplexes these resources to different applications based on their requests. Due to the increasing amount of parallelism in modern multicore processors, there could be more applications executing in the system at the same time. This means that more resources are shared in the system.

In multicore processors, the memory subsystem is generally where the sharing increases the most since each core accesses memory through this subsystem. As a result, this subsystem has become more and more complex and sharing happens at different levels for different cores. For example, cores share different levels of cache or they might access different memory locations through different memory controllers using different interconnect links. NUMA as mentioned before is a result of memory access through different possible paths to memory. To understand how the memory subsystem of multicore systems looks like and what units share which parts of it, we need to take a look at current multicore architectures.

Throughout this chapter, by node we mean a NUMA node and each NUMA node by definition has a separate memory controller. In section 3.1, we first look at two modern multicore processors. In section 3.2, we discuss the mechanisms for providing memory consistency model across multicore processors and their possible effects on performance isolation. We also look at how STREAM benchmark performs on the multicore processors of section 3.1 at the end of this chapter.

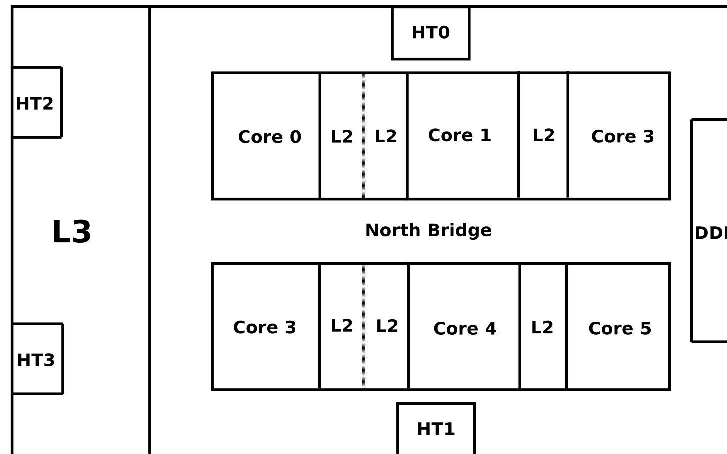


FIGURE 3.1: One of Magny-Cours' dies.

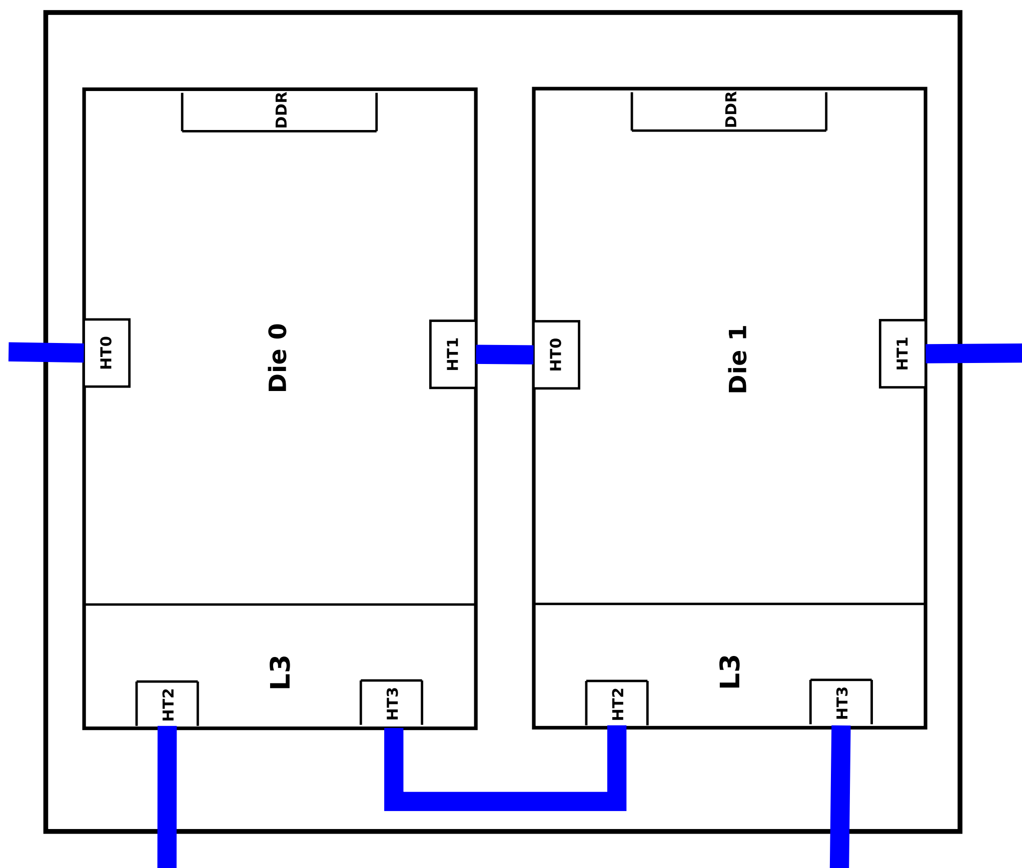


FIGURE 3.2: Magny-Cours processor architecture. The blue lines are hypertransport links. This figure shows how the hypertransport links are used to connect two dies of a processor together.

## 3.1 Current multicore architectures

We take a look at two modern multicore architectures, one by AMD and one by Intel. The important aspects for sharing which we take a closer look at are:

- Cores sharing caches
- Hardware threading
- Memory controllers
- Crossbar switch which routes interprocessor data.
- Interconnect topology

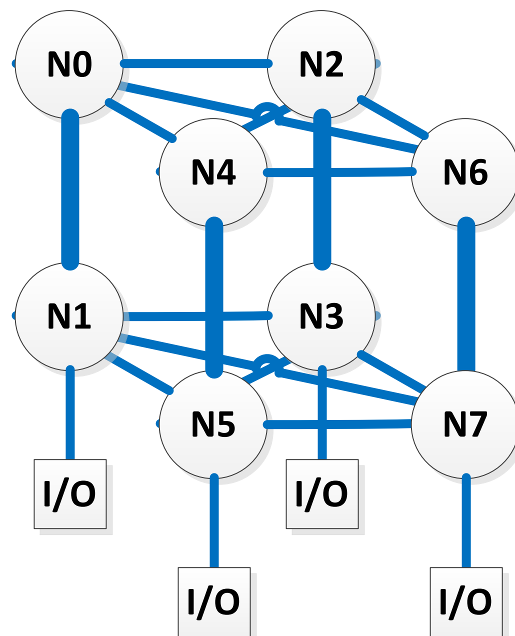


FIGURE 3.3: Interconnect topology of a machine with four Magny-Cours processors. $N_i$ is referred to node $i$. Each two dies form a processor and since each has a memory controller, each processor has two NUMA nodes. $N_0$ and $N_1$ are nodes of the first processor, $N_2$ and $N_3$ are nodes of the second processor and so on.

### 3.1.1 AMD Magny-Cours

Introduced in 2009, AMD Magny-Cours is a multi-chip module (MCM) with twelve cores. This processor has two dies, each with six cores. Each core has a separate unified 512 KB L2 and all the cores in the same die share 1) 6 MB of L3 cache and 2) a memory controller with two channels. Thus, the first level of sharing happens at the L3 cache and then the memory controller. Figure 3.1 shows the architecture of each die. With this architecture, the cores of each die become a NUMA group since they all have the same access latency to the memory attached to their memory controller. One other

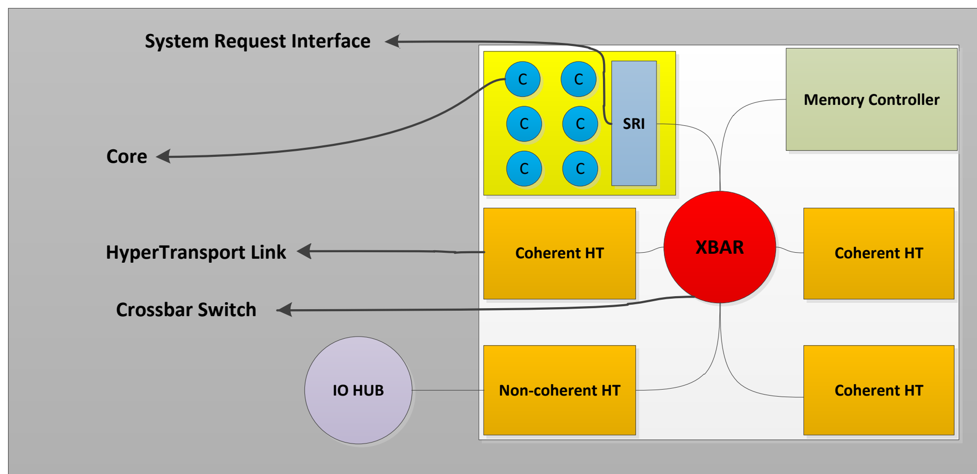


Figure 3.4: Magny-Cours crossbar switch which routes data between different die's components.

observation is that each die has four hypertransport links. As shown in figure 3.2, two of these links are used within the processor: one reads from the other die's memory and second one reads from the other die's L3 cache. With this architecture, each die will have two free hypertransport link and thus the total number of free links in the processor package is four. Each of these hypertrasnport links are divided into to two sublinks which can operate separately and provide eight connections to outside processors or I/O hubs. For more information on the architecture of Magny-Cours, one can refer to [34].

One possible multiprocessor topology with Magny-Cours processors is 4P which is shown in figure 3.3. In this topology, nodes are connected with maximum link distance of two and nodes and I/O hubs with maximum link distance of three. As a result, hypertransport links are shared when accessing remote memory or when I/O devices transfer DMA memory to/from remote nodes.

Inside each die of Magny-Cours, the coherency messages and data, are routed between the system request interface (SRI), memory controller and hypertransport links via a crossbar switch. The architecture of crossbar switch is shown in figure 3.4. If the requested memory is not cached, then a request is sent from SRI to the crossbar switch which in turn is forwarded to the local memory controller or one of coherent hypertransport links depending on the location of memory. Depending on the architecture, sometimes it is required to also send coherency messages on the interconnect when accessing uncached memory. We will talk more about coherency messages in section 3.2 of this chapter.

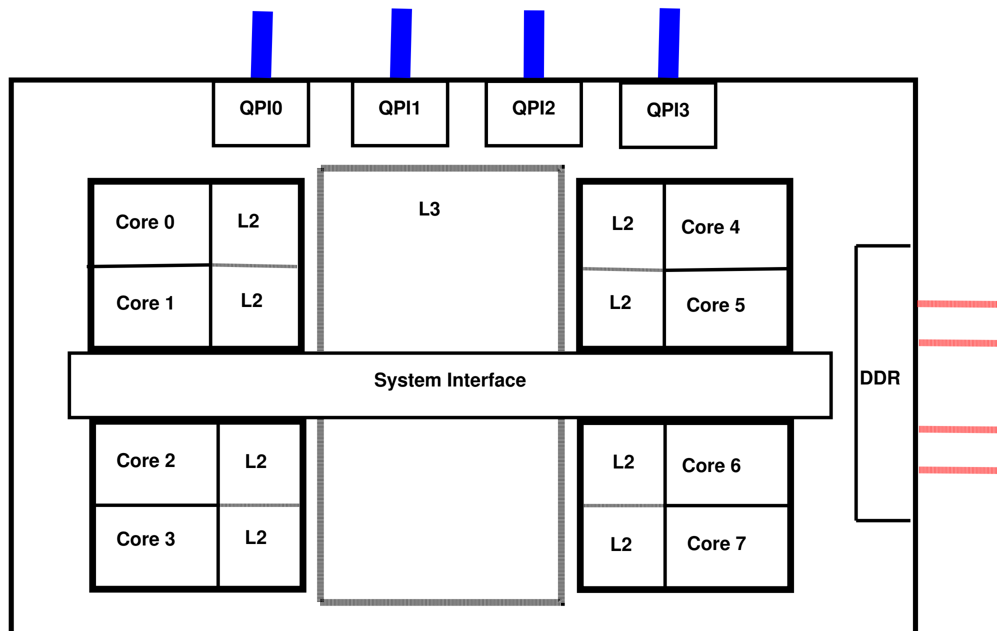


FIGURE 3.5: Intel Nehalem-EX processor die.

In chapter 4, we will examine under which conditions, if any, the memory controller, the hypertransport links and the crossbar switch become the performance bottleneck.

### 3.1.2 Intel Nehalem-EX

Intel Nehalem-EX is a processor released in 2010 with eight cores in a single die (figure 3.5). Each core has 256 KB of L2 cache and cores of each processor share a 24 MB L3 cache. Nehalem-EX supports Simultaneous MultiThreading (SMT) and for each of those cores, two hardware threads exist. As a result, there are sixteen hardware threads available per processor. Each of these hardware threads share a core's different units. Whenever one of the core's thread is blocking for memory or any other event, the other thread can use the core's units and thus the total utilization of the core is increased. There is a hardware scheduler which schedules the hardware threads of a core. Obviously, hardware threads share many resources of a core including all levels of cache.

The onchip memory controller of Nehalem-EX, supports four memory channels shared between all of its eight cores. Intel uses QuickPath Interconnect (QPI), a similar technology to hypertransport, to enable inter-communication channels between processors. One possible 8P topology with Nehalem-EX processors is in figure 3.6. In this topology, processors may share QPI links to access remote memory or when interacting with I/O devices. The maximum interconnect link distance is two, both for accessing remote memory or interaction with I/O devices.

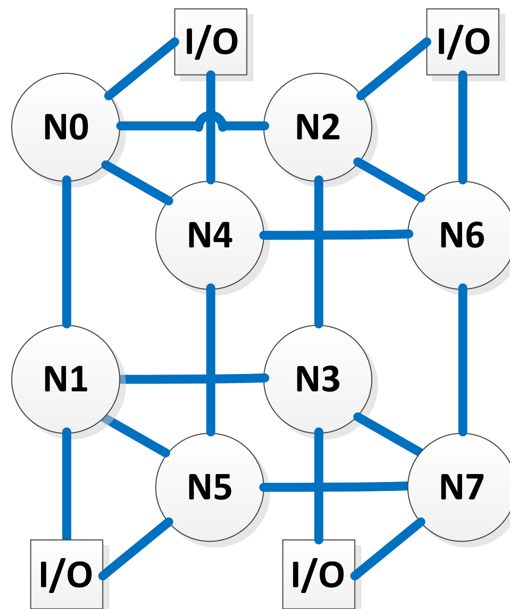


FIGURE 3.6: Interconnect topology of a machine with eight Nehalem-EX processors.

There is not much detail available on the crossbar switch architecture of Nehalem-EX. We assume that it is similar to that of Magny-Cours.

## 3.2 Cache coherency and cache directory

"A memory consistency model for a shared-memory multiprocessor [or multicore] specifies how memory behaves with respect to read and write operations from multiple processors" [35]. A cache coherency protocol implements the memory consistency model of a multicore system. Cache coherency protocol implements the consistency model by sending *coherency messages* on the interconnect when necessary. There are many cache coherency protocols for different consistency models provided by different processors. They are usually thoroughly explored in the manual of every modern multicore processor and studying these protocols is out of the scope of this thesis. Other than added latency when operating on local memory, these messages consume interconnect bandwidth.

Cache directory is a mechanism developed to reduce the amount of traffic caused by cache-coherency protocol on interconnect links. Depending on the architecture, sometimes each processor (or die) has a cache directory in which it tracks whether a cacheline from local memory is cached in any of the other processors in the system. This greatly

reduces the traffic on the interconnect because when a processor accesses its local memory since it does not need to check with all the processors in the system to see whether they hold a copy of that memory; It simply checks its local cache directory. We expect that a cache directory improves the latency of accessing local memory and also improves the whole system performance by reducing the traffic on the interconnect. Since not all modern multicore processors have such a mechanism, it is important to know the differences it might make on performance isolation. Thus, in the following chapters we keep this factor in mind when we suspect it might make a difference in our benchmarks.

It is also interesting to note the travel path for coherency messages between processors or dies. In Magny-Cours (look at figure 3.4), coherency messages are generated from the system request interface if necessary, then they are forwarded to the crossbar switch which decides where the messages need to go. Based on this decision, one of the coherent hypertransport links is utilized to transfer the message to the remote node. In the remote node, upon receiving the message from one of the hypertransport links, it is forwarded to the crossbar switch. Crossbar switch decides whether it should route the message to another node or it is meant for this node. If the message should be routed, it forwards the message to one of the other hypertransport links and if it is local, it is routed to the system request interface.

## 3.3 STREAM on modern multicore systems

STREAM [20] is a famous benchmark used in the processor market to advertise the amount of memory bandwidth of a new processor. This benchmark computes the provided memory bandwidth by running a set of simple operations like copying a large number of array elements. Although it can run on multiple cores at once and measure the total system memory bandwidth, we run it in single threaded mode to measure the memory bandwidth of one of the memory controllers in the system.

We use two testing machines throughout this thesis. One consists of four Magny-Cours processors running at 2.2 GHz with topology shown in figure 3.3 and the other consists of eight Nehalem-EX processors running at 1.8 GHz with topology shown in figure 3.6. The STREAM results for these machines running Linux kernel 2.6.32 are in table 3.1.

| Testing Machine | Copy (MB/s) | Scale (MB/s) | Add (MB/s) | Triad (MB/s) |
|---|---|---|---|---|
| 4P Magny-Cours | 5949.1037 | 5834.0315 | 5789.5724 | 6086.6037 |
| 8P Nehalem-EX | 5344.7646 | 5007.7505 | 5795.0719 | 5583.2550 |

Table 3.1: STREAM benchmark rating for a 4P Magny-Cours machine and an 8P Nehalem-EX machine.

It is possible to turn off Magny-Cours' cache directory. As discussed in the previous section, we expect to see a lower performance without cache directory in a multiprocessor system. Table 3.2, which shows the STREAM ratings with disabled cache directory verifies this.

| Testing Machine | Copy (MB/s) | Scale (MB/s) | Add (MB/s) | Triad (MB/s) |
|---|---|---|---|---|
| 4P Magny-Cours | 3122.2138 | 3125.3400 | 2957.8578 | 3115.8837 |

Table 3.2: STREAM benchmark rating for a 4P Magny-Cours machine when cache directory is disabled.

In section 4.2, we show that our synthetic microbenchmark (described in section 4.1) is compatible with STREAM.

## 3.4 Summary

In this chapter, we discussed the architecture of two modern systems to discover potential sharing which can result in reduced performance isolation. The following resources are shared in modern multicore systems:

- hardware threads share core's cycles and all memory resources.
- Memory controller is shared between cores accessing memory through it.
- Cores depending on the architecture share cache. In most of the current modern processors L3 is usually shared.
- Cores of the same die or processor share their crossbar switch which routes data or coherency messages interdie or interprocessor.
- Interconnect links between processors are sometimes shared depending on the architecture.
- Accessing memory of remote nodes will increase interconnect traffic. If a cache directory is not present, accessing local memory also generates interconnect traffic.

We also looked at the ratings of the STREAM benchmark running on the multicore processors we described in this chapter.

# Chapter 4

# Resource sharing and performance in multicore systems

To assess the amount of influence on performance when using shared resources in a multicore system, we use a synthetic microbenchmark. For each of the shared resources in the system, we run this benchmark in different scenarios to show the effect of sharing.

In section 4.1, we describe our synthetic microbenchmark and in section 4.1.2, our testing environment. Section 4.1.3 explains how to read various experiment setup diagrams. We look at the effects on performance when sharing the main memory bandwidth in section 4.2, sharing the crossbar switch in section 4.3.1 and 4.3.2, sharing the interconnect links in section 4.3.3 and hardware threads sharing a core's resources in section 4.4. Based on these observations, in section 4.5, we summarize the concrete conditions under which performance isolation can be provided by the system.

We do not analyze the pollution over shared caches in this chapter. This effect in multicore processors was recently analyzed by Lee et al. [14] in the context of database systems. They show different performance degradations based on the memory access patterns of the cores with a shared cache. In section 5.3 of this thesis, we compare the relative amount of performance degradation caused by shared cache and other memory subsystems versus performance degradation caused by sharing the memory bandwidth.

## 4.1 A synthetic microbenchmark

We describe a simple synthetic microbenchmark to measure the amount of memory bandwidth observed by cores when putting load on different parts of the memory subsystem. It consists of different parts to provide flexibility for running it in different scenarios.

**Measuring thread** This thread which is usually placed only on one execution context (core or hardware thread) measures the amount of time it takes to read certain amount of memory.

Here is the source of the main part of the measuring thread:

```
    for(int i = 0; i < iterations; i++) {

        start = read_timestamp();
        for(int j = 0; j < size; j += stride) {
            final += ((char*)data)[j];
        }
        stop = read_timestamp();

        results[i] = (stop - start) - cost_of_reading_timestamp;
    }
```

Listing 4.1: Core of the measuring thread

`size` in line four can be varied so that it is bigger than the L3 cache size[1], insuring that the data array does not fit in the cache. If we make sure that the array does not fit into the cache, then each memory access will result into a cache miss and data is fetched from memory. This way, we avoid the performance effects of shared caches.

`stride` in line four is varied to change the number of requests sent to the memory controller per unit time. We always set it to cachline[2] size, to make sure that each access creates a cache miss. This results into the maximum memory load that this microbenchmark can provide. We verified that each memory access results into a cache miss when `stride` is set to cacheline size using the L1 data cache miss ratio of a core which is running the measuring thread. The L1 data cache (L1D) miss ratio is calculated using formula 4.1 on Magny-Cours [27]. The measured `Refills from L2` is always zero for this microbenchmark since the size of the array is larger than L2. Refills from northbridge can either come from L3, another processor cache or DRAM [32]. For this microbenchmark refills can only come from DRAM since memory is allocated locally and the measured L3 hitratio is near zero, meaning L1 misses are not refilled from L3.

$$\text{L1D missratio} = \frac{\text{Refills from northbridge} + \text{Refills from L2}}{\text{L1D requests}} \tag{4.1}$$

[1] L3 is the last level cache in our testing machines.

[2] Cacheline is the smallest unit of memory where system maintains coherency and is usually 32 or 64 bytes.

The formula above gives a value close to one, which means that each access by the microbenchmark results into a cache miss and will result in a cacheline transfer from main memory.

We store the execution time of the `for` loop over the array for a number of iterations. We explain how we use this information to calculate the memory bandwidth in section 4.1.1. Whenever relevant, we also report the bandwidth reported by STREAM to compare the compatibility of our microbenchmark with it.

**Load generating threads** We use these threads to increase the consumption of the shared memory bandwidth.

These threads run a similar `for` loop with two differences compared with the measuring thread:

1. They do not measure the time it takes for the `for` loop to execute, reducing the time window given by the slow instruction which reads the timestamp register and also they do not write the results to memory for the same purpose.
2. The `for` loop is repeated indefinitely instead of a limited number of iterations to avoid cool down phase when the measuring thread is running.

We run the load generating threads on different cores than the core which is running the measuring thread. Depending on the purpose of the experiment they might be placed on different cores and access memory of different memory controllers. We explain in each experiment where they are placed and which memory they access.

**Management thread** This thread is in control of the following:

- Experiment setup: It instructs which execution context should run the measurement thread, which ones should run the load generating threads and which threads should access which memory locations.
- Synchronization: It makes sure that the measuring thread starts measurement only after all load generating threads are running.
- Results: It stores the execution times reported by the measuring thread into a file.

### 4.1.1 Processing the results

As described, the measuring thread measures the execution time of the `for` loop over the array for a number of iterations. The way the load generating threads work avoids cool down phase. If we discard the first warm up iterations then we are always measuring the execution time in the steady state. In this benchmark running on Barrelfish, only the first iteration is in warm up phase which brings in the page table entries for the array into the TLB cache.

With the execution times in the steady state, we extract the median, minimum and maximum of the execution time of the `for` loop over the array. The execution times as mentioned are in cycles. The bandwidth is then calculated using formula 4.2:

$$\text{Memory bandwidth (MB/s)} = \frac{\text{size in MB}}{\text{cycle_to_second(execution time)}} \tag{4.2}$$

Note that `cycle_to_second` is processor specific and depends on the clock speed of the processor. In Barrelfish, there is no functionality to change the default clock speed and thus we do not need to worry about varying core clocks.

Whenever the difference between minimum and maximum in the observed bandwidth is significant, we elaborate more.

### 4.1.2 Experiment environment

Most of the experiments run on a 48-core machine consisted of Magny-Cours processors in 4P topology as discussed in the previous chapter. Memory modules are installed for each processor with 1333 MHz DDR3. The other testing machine is a Nehalem-EX based system containing 32 physical cores and 64 hardware threads in 8P topology shown in figure 3.6. Both machines have cacheline size of 64 bytes.

For each experiment, we mention which testing machine we use. For most of the experiments, we use the 48-core machine because it lets us turn off dies' cache directories in its BIOS.

As mentioned in the previous chapter, we use node interchangeably with NUMA node. Since Magny-Cours has two dies with two different memory controller, each processor has two nodes whereas Nehalem-EX is a single die with a single memory controller and as a result, it is a single node processor.

### 4.1.3 Reading experiment setup diagrams

We explain how to interpret the experiment setup diagrams throughout this chapter. One such a diagram is in figure 4.1. In such diagrams, M stands for the measuring thread and L for the load generating threads. The core which is running the measuring thread is blue, the cores which are running the load generating threads are red and the idle cores are yellow. The gray zone shows the processor package and the lighter gray shows the die(s).

## 4.2 Cores sharing a memory controller

We know that cores can share the memory bandwidth of a memory controller. In this experiment, we are trying to understand to what extent it is possible to share the memory controller provided bandwidth without performance degradation.

**Testing machine:** Magny-Cours. Since we know that each die in a processor has a dedicated memory controller, we vary the number of cores accessing local memory in the same die. The memory controller has two DDR3 channels running at 1333 MHz. Magny-Cours can use 1 MB of each of its nodes' 6 MB L3 cache for cache directory.

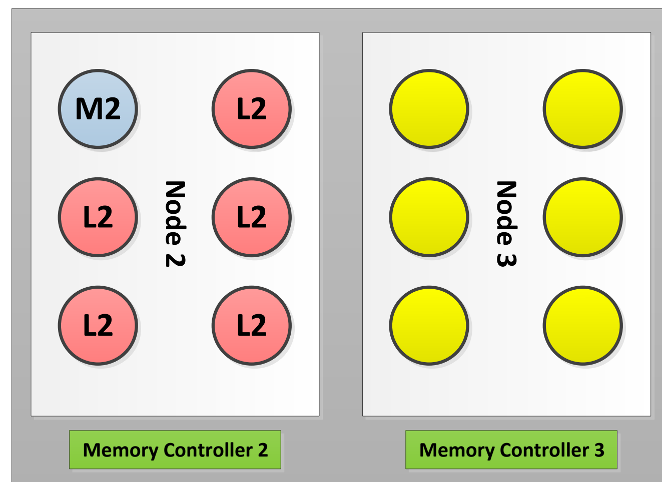


FIGURE 4.1: The experiment setup for the memory controller experiment.

**Experiment setup:** The measuring thread runs on one core and we iteratively increase the number of load generating threads running on different cores of the same die. All the threads access local memory. The experiment is repeated once with cache

directory and once without cache directory by disabling it. The setup sketch of this experiment is in figure 4.1.

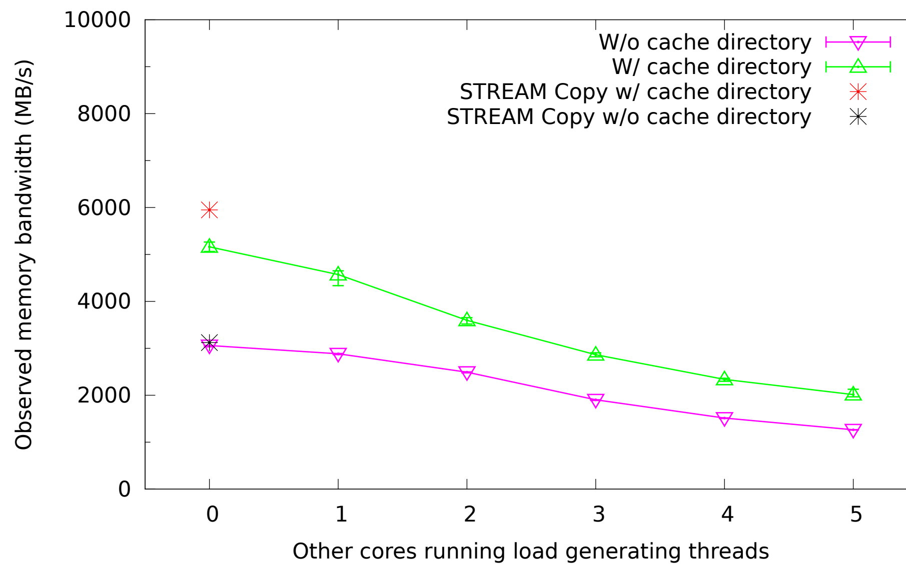


FIGURE 4.2: Performance degradation caused by sharing the main memory bandwidth.

**Discussion:** Figure 4.2 shows the result. With cache directory, just by adding one load generating thread the observed memory bandwidth by the measuring thread drops by 11%. When all 5 load generating threads are enabled, the observed bandwidth drops by 61%. Similar effect happens when we disable the cache directory. *This experiment shows that we need to control the consumption of the shared bandwidth to provide performance isolation.*

It is interesting to see that in Magny-Cours only two cores can see the effect on performance when using the shared bandwidth. As mentioned before in section 2.1.3, current memory controllers do not provide a facility to control the consumption of the shared bandwidth and as a result, we need a fully software-based approach to do this. We explain a mechanism that we use to control the shared memory bandwidth in section 5.2.1.

Figure 4.2 also includes the Copy bandwidth from STREAM and we can see that our microbenchmark reaches almost the same memory bandwidth. One of our observations in section 6.2.1 was that different NUMA groups see different memory bandwidth or latency. Thus, sometimes the measured bandwidth of our microbenchmark is closer to STREAM as we place our measuring thread on different NUMA nodes. We include STREAM ratings whenever they are relevant to provide a basis for comparison.

## 4.3 Effects of the shared interconnect

One important question for performance isolation is the performance effects of traffic on the shared interconnect. There are different scenarios where processor units like the crossbar switch or the interconnect links are shared between cores. In this section, we experiment with these scenarios to see in what situations they become the performance bottleneck.

**Testing machine:** Magny-Cours. This machine has four processors, each having two dies with different memory controllers resulting in eight NUMA groups. Magny-Cours uses 1 MB of each of its nodes' 6 MB L3 cache for cache directory. As shown in figure 3.3, nodes are connected with each other via either one hyper transport link or two. A middle node is also active in the transmission of data between the two nodes when their hyper transport link distance is two.

### 4.3.1 Performance degradation caused by coherency messages

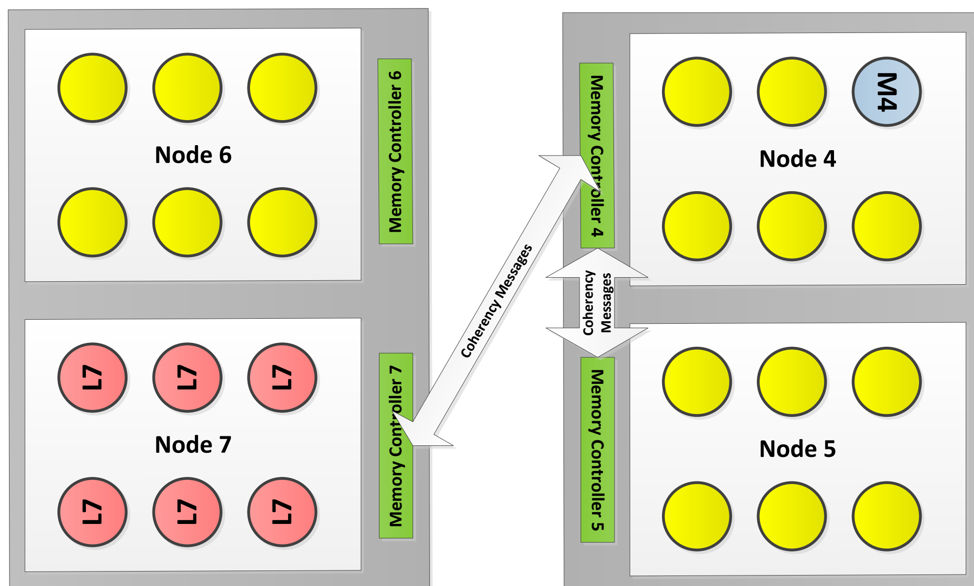


FIGURE 4.3: Experiment setup 1 diagram. The gray arrows represent hyper transport links. The coherency broadcasts of node seven is routed by node four to node five.

From the architectural description of Magny-Cours, we know that when two nodes are located within a single processor, one of their connecting hyper transport links is used just for transferring caches. In our experiment threads do not share any memory. Thus, if the nodes are directly connected, there is no difference between two nodes located in a single processor or different processors.

As mentioned, in our testing machine it is possible that one node routes memory or coherency messages between two nodes. We experiment with performance indications of routing memory or coherency messages on the performance of local memory of the node which does the routing for another node.

The information regarding routing of the data or coherency messages can be obtained by reading the PCI configuration space. For more information one can consult with Magny-Cours' manual [32].

**Experiment setup 1:** First, we experiment with indications of routing coherency messages. We run one measuring thread on a node and we start running the load generating threads on another node. The measuring thread's node has to route the coherency messages of load generating threads' node. We expect to see a difference with cache directory since it greatly reduces the coherency messages that the dies need to send on the interconnect. Thus we experiment once with and once without cache directory. The setup sketch of this experiment is in figure 4.3.

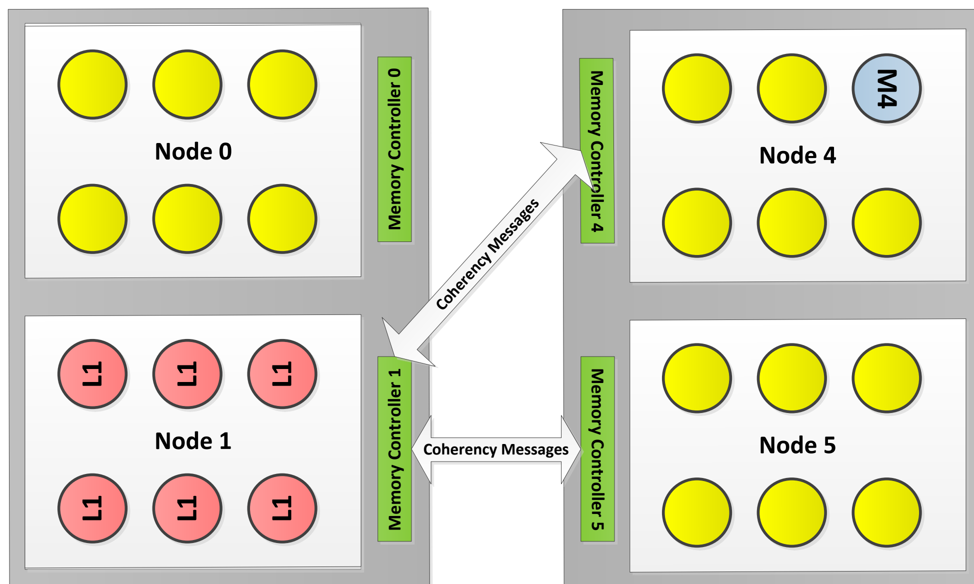


FIGURE 4.4: Experiment setup 2 diagram. Coherency broadcasts of node one is transferred directly to both nodes four and five and no routing is necessary by node four.

**Experiment setup 2:** Same as setup 1, except that the node which is running the measuring threads does not have to route the coherency messages of the node which is running the load generating threads. The setup sketch of this experiment is in figure 4.4.

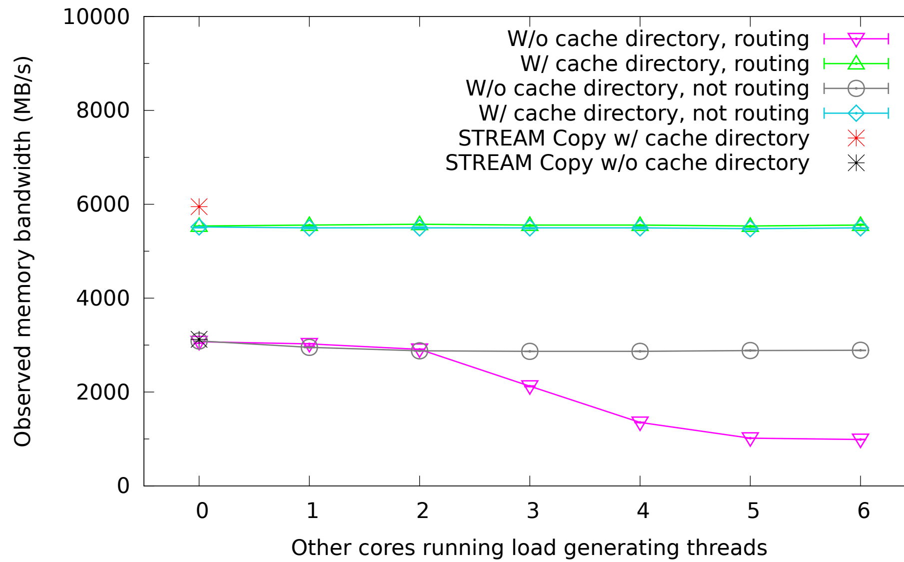


FIGURE 4.5: Performance degradation caused by coherency traffic.

**Discussion:** Figure 4.5 shows the result. With cache directory, the system shows no performance degradation. This is expected since accessing local memory should not result in coherency messages and as a result load generating threads do not generate any (or minimal) coherency messages. However, when cache directory is disabled load generating threads do generate a lot of coherency messages. When the measuring thread's node does not have to route the coherency messages, the observed memory bandwidth drops by 7% and when it also needs to route the coherency messages by 68%. According to these experiments, *whenever there is no cache directory in the system, it is not possible to provide complete memory isolation unless some methods are provided to limit the possible number of coherency messages on the interconnect.*

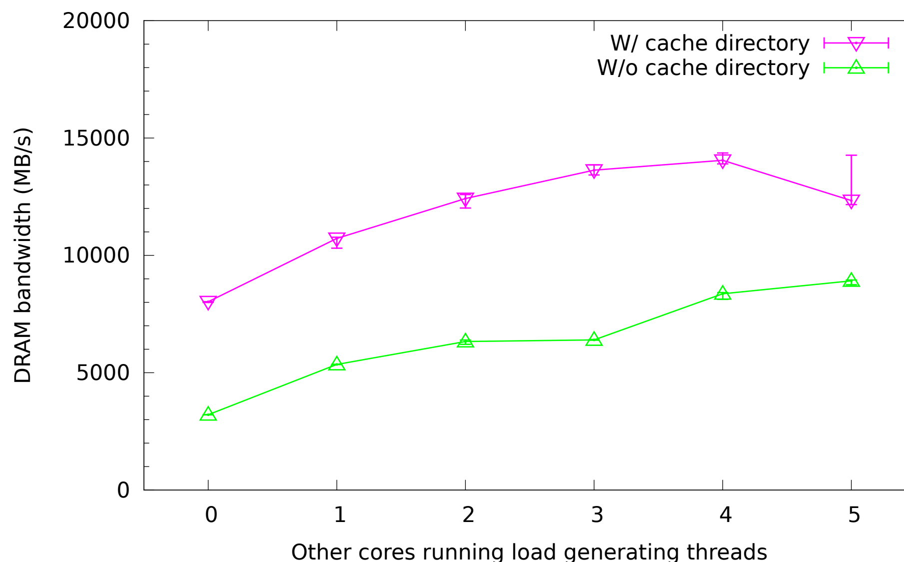


FIGURE 4.6: Performance degradation caused by coherency traffic.

Magny-Cours' memory architecture [36] routes remote coherency messages or data through the crossbar switch. As shown in figure 3.4, cores which access local memory share the crossbar switch with interconnect links. These experiments show that crossbar switch is a potential performance bottleneck when routing coherency messages.

The result of this experiment is very interesting for the case with disabled cache directory. As we there more than two load generating threads, the coherency traffic caused by their memory bandwidth consumption, becomes a bottleneck for a remote node. Figure 4.6 shows the consumed DRAM bandwidth of our microbenchmark. When the consumed local memory bandwidth goes above 6.3 GB/s, the coherency traffic creates a bottleneck on the remote processors. The DRAM bandwidth is calculated using formula 4.3 [27].

$$\text{DRAM bandwidth (MB/s)} = \frac{(\text{DRAM0 events} + \text{DRAM1 events}) \times 64}{\text{Time in second} \times (1024 \times 1024)} \tag{4.3}$$

It is also interesting to note that with cache directory, we see a considerable decrease in performance when adding the last load generating thread. We speculate that it has something to do with the number of cores that are sending requests to the system request interface (see diagram 3.4) since with five cores we already see a higher DRAM bandwidth and the crossbar switch is not concerned with the cores.

Now the question is to what extent it is possible to increase the load on the memory subsystem without observing performance degradation caused by coherency messages *with* cache directory.

**Experiment setup 3:** We run our measuring thread on one of the nodes and start running the load generating threads on cores of the other nodes in the system. We have eight nodes (each with separate memory controller) in the system each with six cores. We thus have the maximum of 42 cores running the load generating threads. Figure 4.7 shows the experiment setup's sketch.

**Discussion:** As we can see in figure 4.8, with cache directory there is no performance degradation in the system. This interesting results suggest that *firewalling from other processors in the system is not necessary when a processor is running in isolation with a cache directory mechanism in place. This is due to minimized cache coherency traffic on the interconnect.*

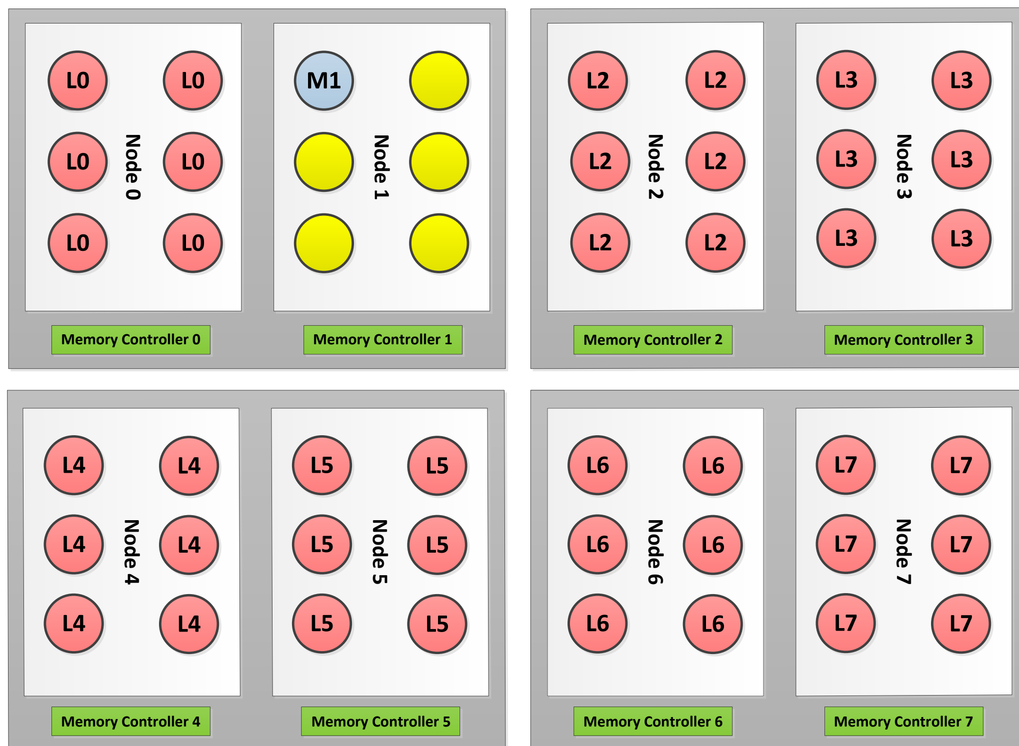


FIGURE 4.7: Experiment setup 3 diagram.

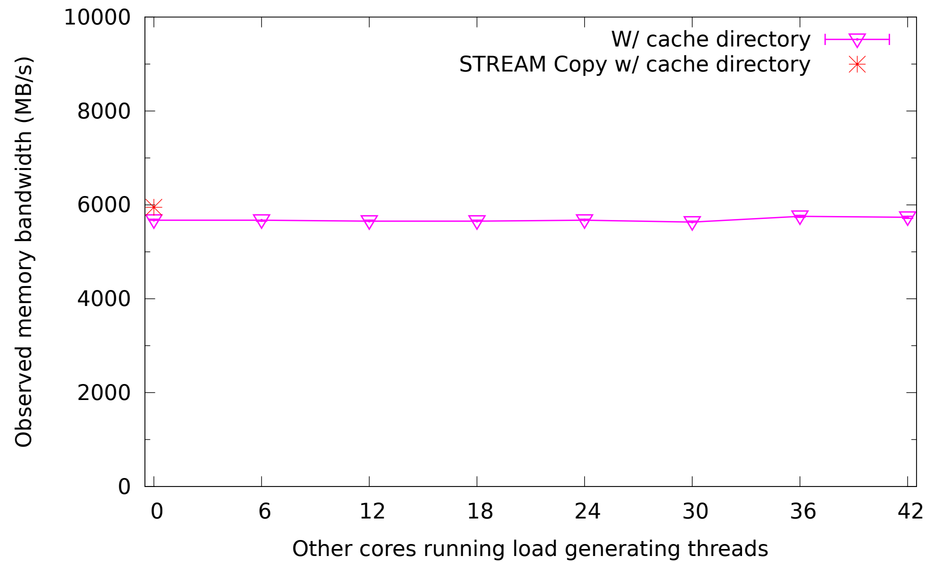


FIGURE 4.8: Performance degradation caused by running load generating threads on all cores.

This suggests that as coherency messages become scalable with cache directory, the memory subsystem does not become the performance bottleneck if all the cores in the system allocate memory locally.

Note that cache directory does not change the fact that accessing remote memory

results in producing coherency messages, but not as much as a system without it. With cache directory, the information on who holds the most recent data is stored in the cache directory and instead of broadcasting coherency messages for the fresh data, the owner of the cached memory can directly send a coherency message to the node which holds the most recent version in its cache. P. Conway et al [34] describe more details on Magny-Cours' coherency messages with cache directory.

### 4.3.2 Performance degradation caused by routing data

In this set of experiments, we investigate whether routing data affects the performance in possible scenarios.

There are two possible scenarios where routing data by a middle node can have some effects on the observed performance:

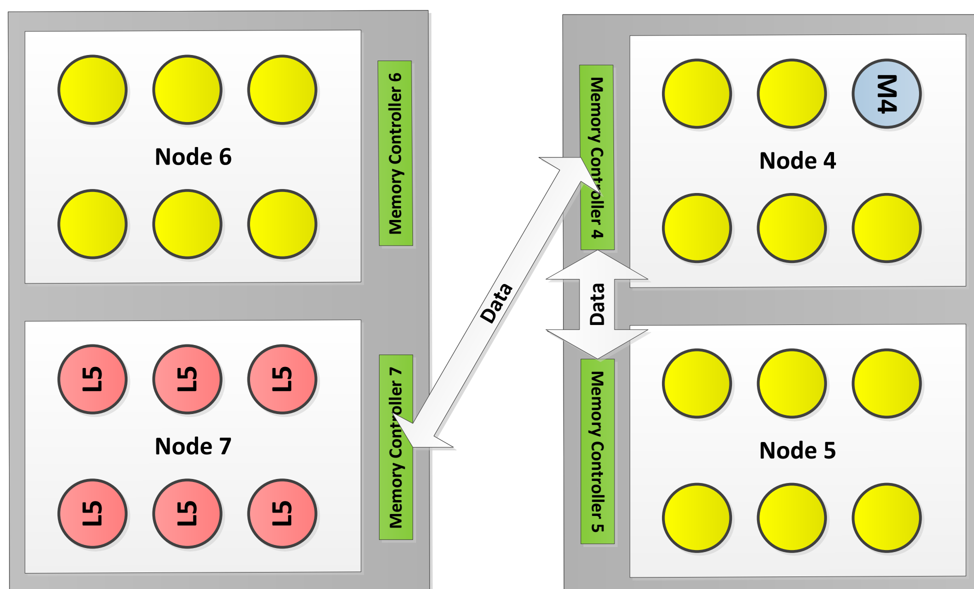


FIGURE 4.9: Experiment setup 4 diagram. Measuring thread is accessing local memory and its node routes the data of load generating threads which are running on a different route. The arrows show how the data is actually routed.

1. Excessive load on the local memory controller of the node which routes the data can affect the performance observed by a remote node.
2. Excessive load sent by remote node on the link can affect the performance of the local memory of a node which routes the data.

**Experiment setup 4:** This experiment is designed to address the first scenario. We run the load generating threads accessing local memory on a node. Then we run our measuring thread, running alone on a different node, accessing remote memory which is routed through the node which is running the load generating threads. We run this experiment with and without cache directory. The sketch of this experiment setup is in figure 4.9.

**Experiment setup 5:** This experiment is designed to address the second scenario. We run the load generating threads in a node and they access remote memory of a node which is routed by another node. Then we run our measuring threads on the routing node accessing local memory. We run this experiment with and without cache directory. The sketch of this experiment setup is in figure 4.10.

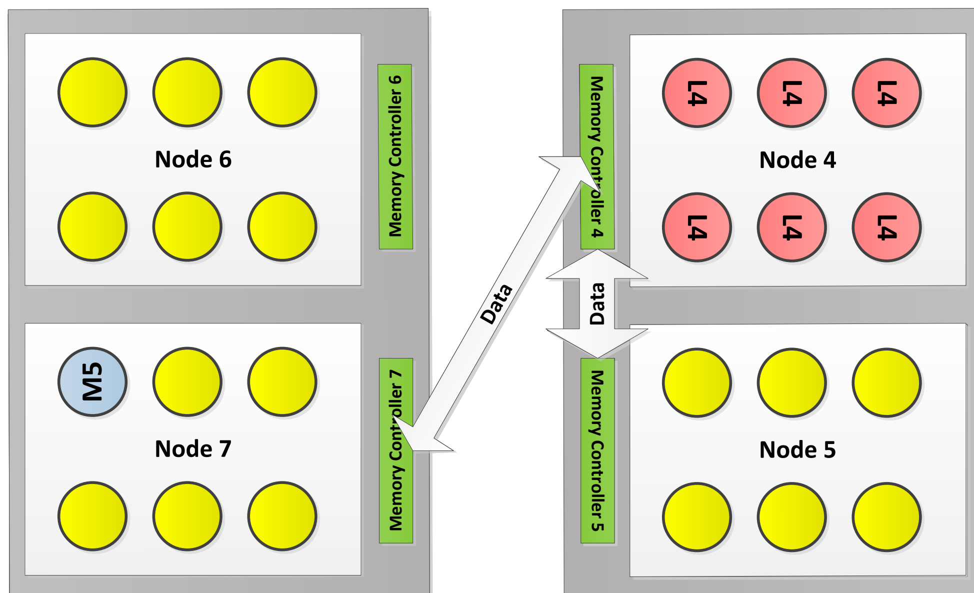


FIGURE 4.10: Experiment setup 5 diagram. Measuring thread is accessing remote memory which is routed by the node which is running load generating threads accessing local memory.

**Discussion:** The results of both experiments are in figure 4.11. These experiments show that crossbar switch is not a performance bottleneck with cache directory even under the load of routing the data of another node. However, again without cache directory, it rapidly becomes a performance bottleneck because of coherency messages observed on the interconnect.

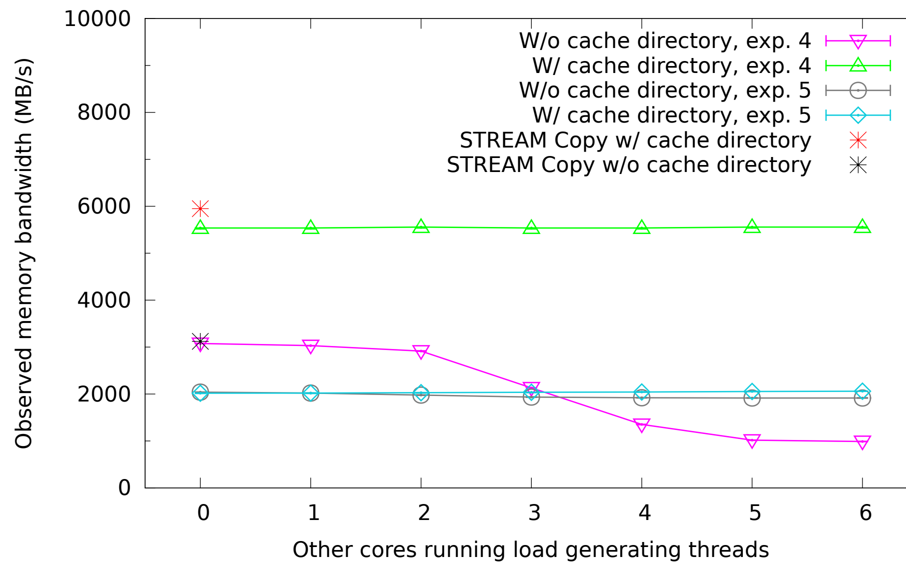


FIGURE 4.11: Performance degradation of different data routing scenarios.

The result of this experiment has an interesting indication on performance isolation: in a cache directory enabled system, we do not need to control the memory bandwidth consumption of a node which routes data.

We can design a more aggressive experiment for cache directory enabled systems which puts more load for data routing where a node routes more than one node's data. However it is quite unlikely that such a scenario happens where nodes are at most two hops away and we move on and focus on more interesting cases.

### 4.3.3 Performance degradation caused by sharing a link

Another interesting case is investigating when the shared interconnect links can potentially become a performance bottleneck. This is important because cores can share them to access the memory of remote nodes. This situation can happen for example when an I/O device is attached to a remote node or when the memory requirement of a process cannot be provided by local memory resources.

**Experiment setup 6:** We run our measuring thread on one of the nodes and start reading remote memory. Then, we iteratively add load generating threads on the cores of the same node reading the memory of the same node as the measuring thread. The remote memory is chosen at a place with only one interconnect link away. The experiment setup's sketch is in figure 4.12. The experiment is repeated by disabling the cache directory.

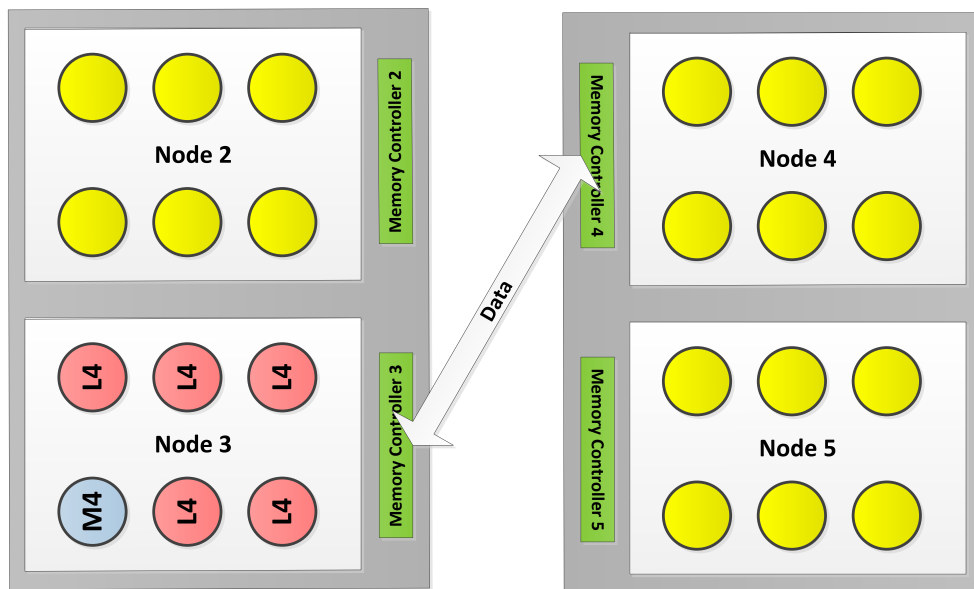


FIGURE 4.12: Experiment setup 6 diagram. Measuring thread and load generating threads run on the same node and access remote memory through the same hyper transport link.

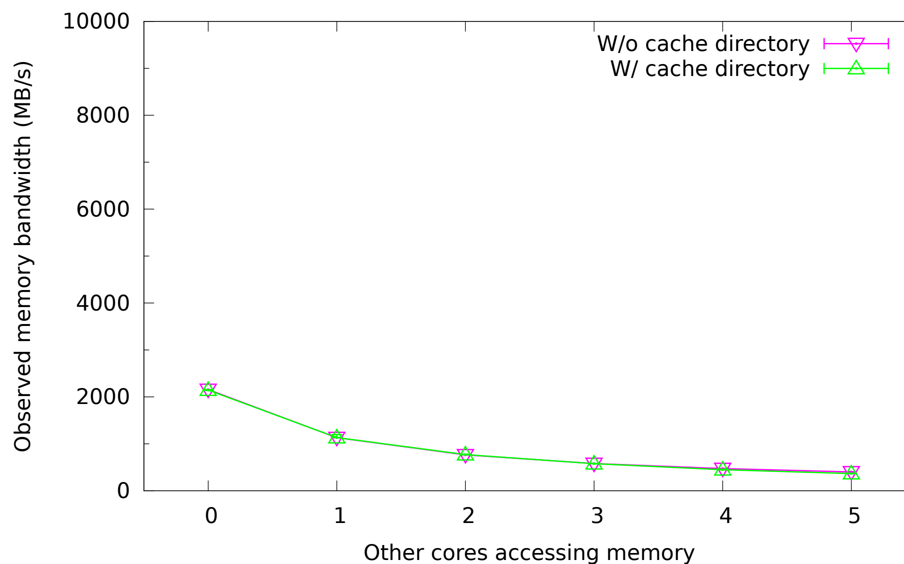


FIGURE 4.13: Performance degradation of sharing an interconnect link.

**Discussion:** Figure 4.13 shows the result of this experiment. Having cache directory does not help to improve performance when accessing remote memory since the cost is dominated by the bulk transfer over the interconnect link. Cache directory cannot improve latency as it does in the local memory controller experiment (figure 4.2) since here we need to wait for the result of cache coherency broadcasts (see page ten of [34]). The memory bandwidth provided by interconnect links of Magny-Cours is slightly over 2 GB/s and it drops to half as soon as another core starts using it. *This experiment*

*shows that since one core can easily saturate the memory bandwidth of an interconnect link, we need to provide a mechanism to isolate these links if a core running in isolation is accessing remote memory through that link.*

In section 6.2.4, we describe an algorithm to automatically extract the interconnect topology and then by using it, we provide a mechanism to isolate core to memory path.

## 4.4 Hardware threads sharing a core

In this experiment, we want to investigate the effects of simultaneous multi threading (SMT) on memory performance.

**Testing machine:** Nehalem-EX. It is chosen because it is the one with SMT support.

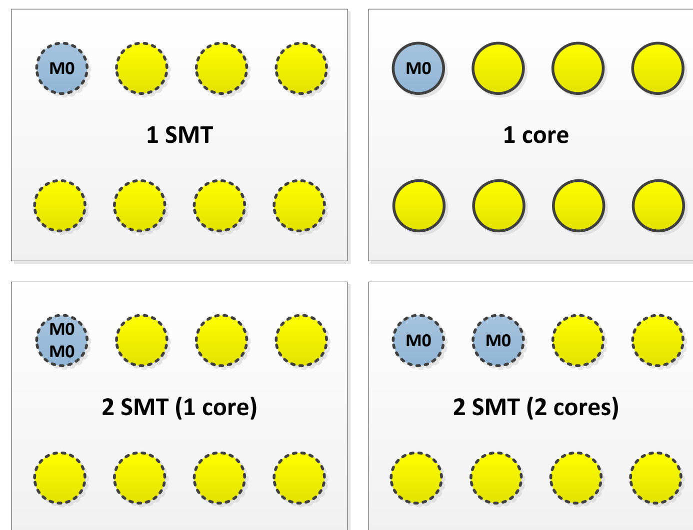


FIGURE 4.14: The SMT experiments. The dashed circles indicate cores with SMT enabled. All setups access local memory of processor 0.

**Experiment setup:** We once execute the measuring thread on one hardware thread. Then on two hardware threads which belong to the same core and then two hardware threads which belong to two different cores. We also turn off SMT and run the same experiment to see the effect of it on performance. All the experiments were executed in one processor and the experiments' setup sketches are in figure 4.14.

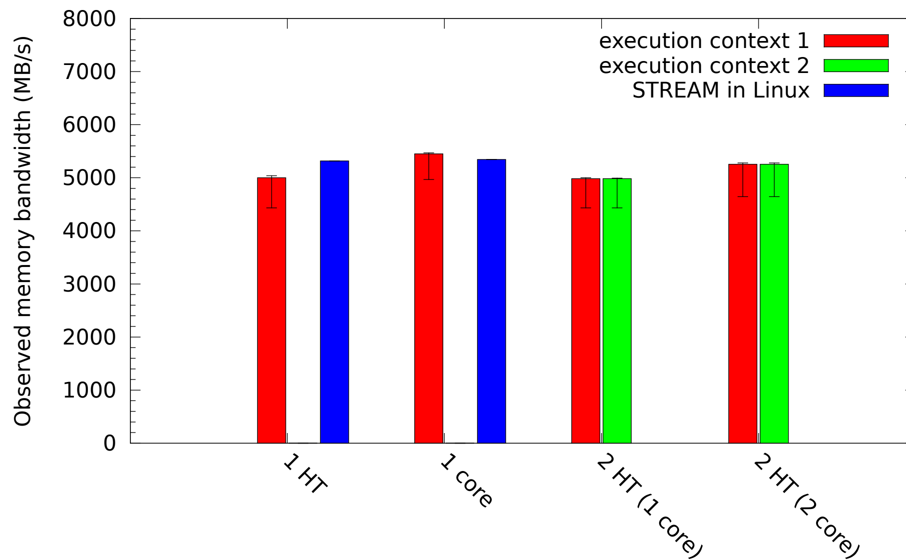


FIGURE 4.15: Performance comparison of various SMT setups.

**Discussion:** Figure 4.15 shows the result. This experiment shows that running two memory intensive processes on the hardware threads of the same core results in performance degradation. Even when we try to run two measuring threads on the hardware threads of different cores, we observe a degradation in each compared to the case where we turn off SMT completely. The reason for this is that the idle hardware thread of the core is running the monitor process (A Barrelfish privileged user process) and the spawn process (a process which is responsible for executing a new process).

The result of this experiment is not very exciting since hardware threads do share a lot of resources of a core including the core's cycles. SMT performs relatively good in this experiment because most of the times hardware threads are blocking for memory and they do not share much of the core's cycles.

However, there are two interesting things which we notice by looking at figure 4.15:

1. STREAM seems not to degrade much when running on a hardware thread in Linux compared to a core. This suggests that Barrelfish uses slightly more resources of the core than Linux due to its distributed nature.

2. One hardware thread (or core) does not affect the observed bandwidth of another hardware thread (or core), because the total consumed memory bandwidth doubles when we use two cores of the same processor. This is not the case for Magny-Cours as shown in figure 4.2.

The measured rather low minimum bandwidth in all the cases, is caused by infrequent peaks which we could not identify due to lack of performance monitoring support in

Barrelfish for this processor. This experiment shows that *if a process with performance isolation guarantees is running on a hardware thread of a core, another process should not be placed on other free hardware threads of this core or SMT should be turned off preferably to remove the noise of other hardware threads which might run other processes or operating system's components.*

## 4.5 Summary

In this chapter we measured different aspects in sharing the memory subsystem in modern multicore systems. We found the following conditions to be necessary for providing performance isolation:

**With SMT** Process allocation on hardware threads should consider isolation aspects. This is due to the fact that most of the resources of a core are shared by its threads. A more aggressive approach for performance isolation would be disabling SMT to minimize the noise caused by other hardware threads of the core.

**With cache directory** The consumption of the provided memory bandwidth by the local memory controller should be controlled if one of the local cores is running in isolation. If a core is accessing remote memory, the *interconnect path* to that memory should be isolated. Since cores usually share the last level cache, to provide perfect performance isolation, when one of the cores is running in performance isolation, no process should run on the cores with the same shared cache.

**Without cache directory** All of the facts for a system with cache directory system also holds here. However, as we saw in figure 4.5, if a node needs to route coherency messages of other cores, the performance of its local memory will degrade. This makes it difficult for a system without cache directory to provide performance isolation since even remote nodes may need to be isolated. This can result in a very low total system utilization.

# Chapter 5

# Performance isolation support in a multicore operating system

In this chapter, we discuss the requirements of supporting performance isolation in a multicore operating system based on the results we obtained in the previous chapter.

If a process[1] wants to run in performance isolation, it needs to provide some information to the operating system regarding its resource requirements. The more detailed information enables the operating system to provide more fine grained performance isolation. The minimal information which operating system might need to know to provide isolation guarantees is the following:

1. The desired location of the isolated core
2. The size of memory
3. The location of memory
4. Required performance properties of the memory

## 5.1 Enforcing performance isolation

We assume that processors provide cache directory or a similar architecture to minimize the interconnect traffic. This is important because as we showed in the previous chapter, performance isolation without cache directory is not feasible in modern multicore systems. Fortunately, most of the recent widely used multicore processors from AMD

[1] A process in Barrelfish context is called a dispatcher.

and Intel provide such a facility. To enforce performance isolation, the operating system needs to ensure the following using the provided information:

1. If there is no SMT support, no other process should run on any of the cores which are running in isolation of a process. With SMT, it is typically not possible to filter out the noise of other hardware threads. Thus, it is better to turn them off to provide performance isolation.
2. If an isolated core is using the shared memory bandwidth of a memory controller, then it should be possible to control consumption of the shared bandwidth of that memory controller in order to avoid performance degradation. For example, each of the memory controllers on the dies of a Magny-Cours provides roughly 6 GB/s of copy bandwidth. We should avoid overcommitting this shared bandwidth by monitoring the amount of memory bandwidth consumption.
3. If an isolated core wishes to access remote memory for any reason, the interconnect path to that memory should be isolated. In our machine with Magny-Cours processors, we measured 2 GB/s of bandwidth (4 GB/s in both directions) in section 4.3.3. This bandwidth is completely saturated by a single core.
4. If an isolated core wishes not to share cache with any other core, it should be able to do so.

By providing more information, an operating system with performance isolation support can perform more fine grained isolation and better resource allocation:

- The isolated amount of bandwidth to memory. We showed in section 4.2 that cores which are sharing the main memory bandwidth can have performance impact on each other. Even running two processes on two different cores which share the memory bandwidth can result in performance degradation of the processes depending on their memory bandwidth requirements. If a process could provide the required amount of bandwidth to memory, then allocating the whole bandwidth of the memory controller to a single core is not necessary. Alternatively, the operating system can monitor the shared memory bandwidth consumption and make sure that the provided memory bandwidth is not overcommitted.
- Information regarding cache sharing. Always isolating L3 cache as we will show in section 7.3, can result in low processor utilization since other cores in the processor which are not isolated are not allowed access to memory and thus cannot run any other process. We will measure the expected amount of degradation when we allow cache sharing in the section 5.3 in this chapter.

- Performance properties of the memory that the process is going to access. For example, asking for a core with lowest latency or highest bandwidth to its NUMA memory. We elaborate more on this point in section 6.2.1.
- A scheduling time slice. If a process provides a time slice in that it needs performance isolation, then it is not necessary to completely allocate the resources to that process. The processes could time multiplex resources. As mentioned before, this idea is currently under research in a multicore operating system called Tessellation which provides time-space partitioning of system resources [10]. The scheduling part of performance isolation is out of the scope of this thesis and we do not discuss it any further.

Except providing memory bandwidth isolation, the other items are directly under control of the operating system and enforcing them is trivial since operating system decides which process runs on which core or which core can access which location in the memory. By controlling these parameters, operating system can easily enforce core isolation, cache isolation and path way to memory isolation (i.e. interconnect links). We discuss possible methods to enforce memory isolation in the next section.

## 5.2 Enforcing memory isolation

According to what we have discussed so far, if it was possible to somehow control the consumption of the shared bandwidth, then it was possible to have higher processor utilization in an operating system with performance isolation support. We can achieve this in two ways:

**Hardware** Using hardware support for dynamically controlling the memory operation rate for a set of cores. Unfortunately, current x86 multicore processors which we looked at do not provide such mechanisms. Thus, we are interested in a software based technique.

**Software** Simulating the required mechanism in software. There are several methods discussed in [3] to do so, but for current multicore processors the only possible way without changing the software binary is by exploiting the performance monitoring counter registers. For this we need to:

1. Measure the actual memory bandwidth provided by the memory controller and shared between different cores. We can do this with a microbenchmark during system start-up. We discuss a simple microbenchmark to do so in section 6.2.1.

2. Dynamically observe the amount of memory bandwidth consumed by cores using the performance counters. If the memory bandwidth is isolated, then overcommitting the bandwidth is not allowed.

Using such a method, we can detect whether the shared bandwidth is overcommited and can hurt the performance of an isolated process. Then, we can either move processes around to different NUMA memory or we can stop allocating memory from that NUMA node with overcommited memory bandwidth. We elaborate more on this in the next section.

### 5.2.1 Memory bandwidth isolation and process migration

After the information regarding current memory bandwidth consumption is available, there are two possible approaches to ensure memory bandwidth isolation:

1. After we detect that the memory bandwidth with isolation guarantees is overcommitted, we should move the processes which are running without isolation to another node so that there is no violation of isolation guarantees. We call the act of moving a process with all of its memory resources to a different node *process migration*.
2. It is possible to take a preventive step and not allocate memory from the memory controller which its bandwidth is nearly overcommited and is isolated.

There are advantages and disadvantages with each approach. The first approach allows for the best possible utilization of memory resources, but process migration is usually costly and not all operating systems provide support for it. The second approach is simpler and there is no need for costly process migration. However, it can lead to under-utilization of memory resources. With the second approach we also need an estimation to decide whether we can consume more memory bandwidth.

Since Barrelfish does not support process migration yet, we decided to go with the second approach. We estimate whether the shared bandwidth will be overcommited after a new allocation using the following algorithm:

```
bool is_bandwidth_available(numa_id numa)
{
    // Current NUMA node bandwidth consumption
    bandwidth_consumption = get_bandwidth_consumption(numa);
    // Maximum available bandwidth
    maximum_bandwidth = get_maximum_bandwidth(numa);
    // Maximum bandwidth consumed by a single core
    core_max_bw = core_max_consumption(numa);

    estimate = bandwidth_consumption + core_max_bw;

    if( estimate > maximum_bandwidth) {
        return true;
    }
    else {
        return false;
    }
}
```

Listing 5.1: Estimation of availability of memory bandwidth for a new core

This algorithm finds the core which is using more memory bandwidth from this NUMA node than the other cores. Then, it checks whether allocation of a new core with the same bandwidth as the most memory bandwidth consuming core will result in overcommitting the bandwidth. We need to do this once for the read bandwidth and once for the write bandwidth since memory controllers usually provide different bandwidth for read and write and applications also have different read or write bandwidth requirement depending on their functionality.

While this (over-)estimation is not the best possible estimation, it is a simple, efficient and sensible one. We leave finding the best possible estimation as an open problem for future work.

## 5.3 Performance degradation caused by sharing L3

In this section, we try to understand the amount of degradation caused by sharing the last level cache. Both of the modern processors which we looked at in chapter 4 have the same property; cores of the same processor (or die) share the last level cache (L3) and as a result, it is of interest for this thesis to measure the amount of performance degradation caused by sharing the last level cache.

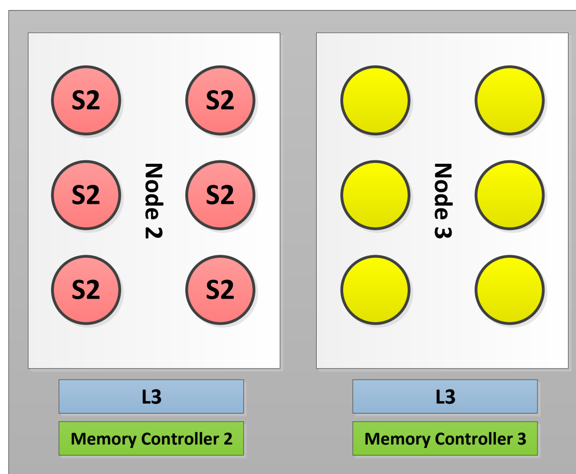


FIGURE 5.1: Experiment setup for shared cache performance degradation when L3 cache and DRAM are shared between benchmark instances. The notation is similar to last chapter. S stands for streamcluster instance. S2 means that this instance is accessing the memory of node two.

It is not trivial to measure the amount of performance degradation caused by sharing L3 and sharing other memory subsystems. This is because L3 and other memory subsystems are connected to each other and the events concerning L3 will affect other subsystems. For example, an L3 cache miss will result in traffic on the memory controller and the processor will also instruct the prefetcher based on this miss. Keeping this in mind, we designed an experiment to roughly estimate the amount of degradation caused by sharing L3 and the other memory subsystems versus the amount of degradation caused by sharing the memory bandwidth.

The key idea in designing this experiment is the fact that accessing remote memory does not cause traffic on the local memory controller.

**Testing machine** 4P Magny-Cours. We take two nodes with distance one for this experiment. Minimal distance is desirable since it results in lowest latency possible when accessing remote memory.

**Experiment setup** We take streamcluster from the PARSEC benchmark suite [37]. More information regarding this benchmark is in section 7.1. We take one of the nodes and add single threaded and synchronized streamcluster instances to its cores accessing local memory. We measure the amount of L3 misses as we add more streamclusters and the time it takes for instances to finish. Only one streamcluster instance reports the statistics. We repeat the benchmark, but this time one instance accesses local memory

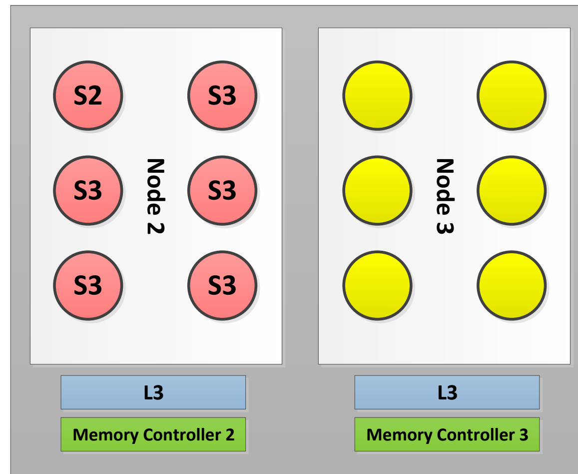


FIGURE 5.2: Experiment setup for shared cache performance degradation when only L3 cache is shared between benchmark instances.

and all the other instances access another node's memory. Only the streamcluster instance which accesses local memory reports the statistics; we call other instances load generating streamcluster instances. In both setups benchmark instances share L3, but in the second experiment the instances which access remote memory do not consume local memory bandwidth. Figure 5.1 shows the first experiment setup and figure 5.2 shows the second experiment setup.

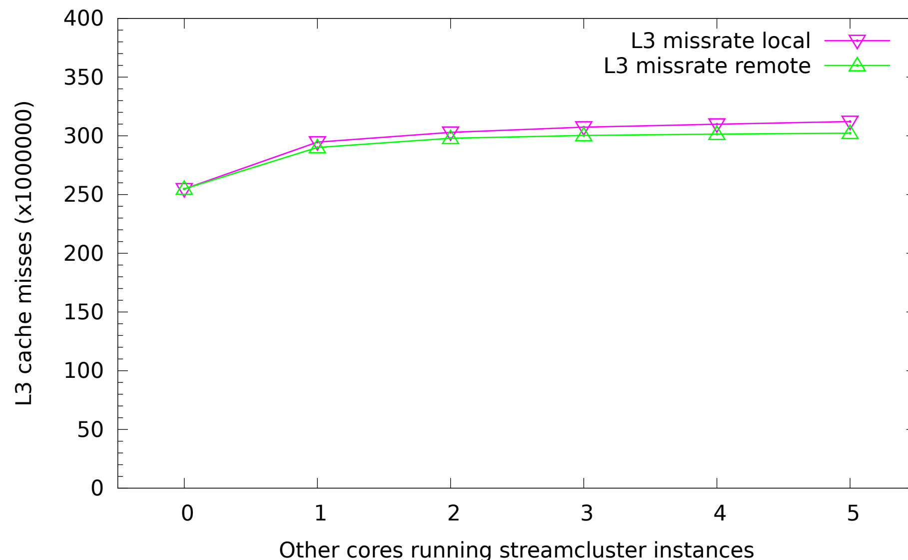


FIGURE 5.3: The comparison of cache miss rate when load generating streamcluster instances access local memory and when they access remote memory.

**Discussion:** Figure 5.3 compares the amount of L3 cache misses (computed with formula 5.1) when load generating streamcluster instances access remote memory instead

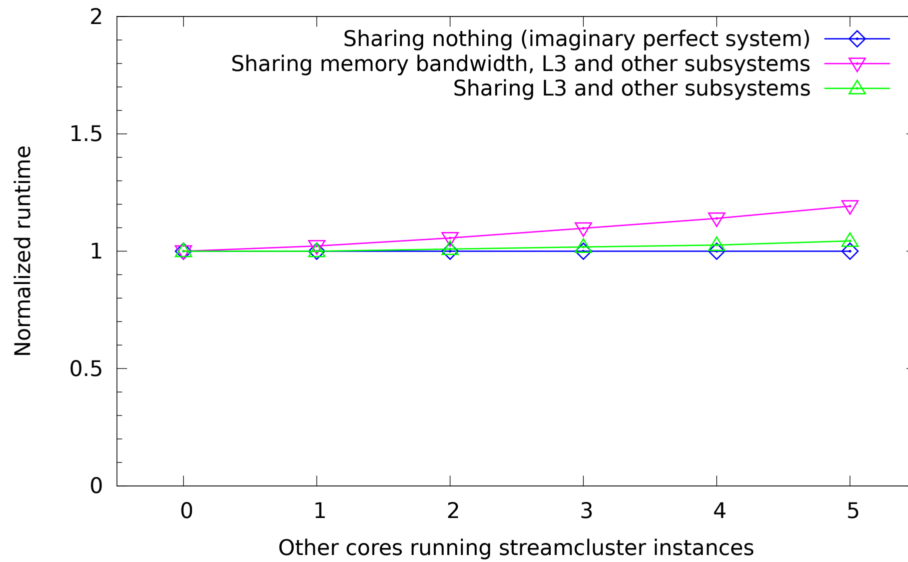


Figure 5.4: The comparison of runtime degradation in two different scenarios when the shared memory bandwidth is shared and when it is not.

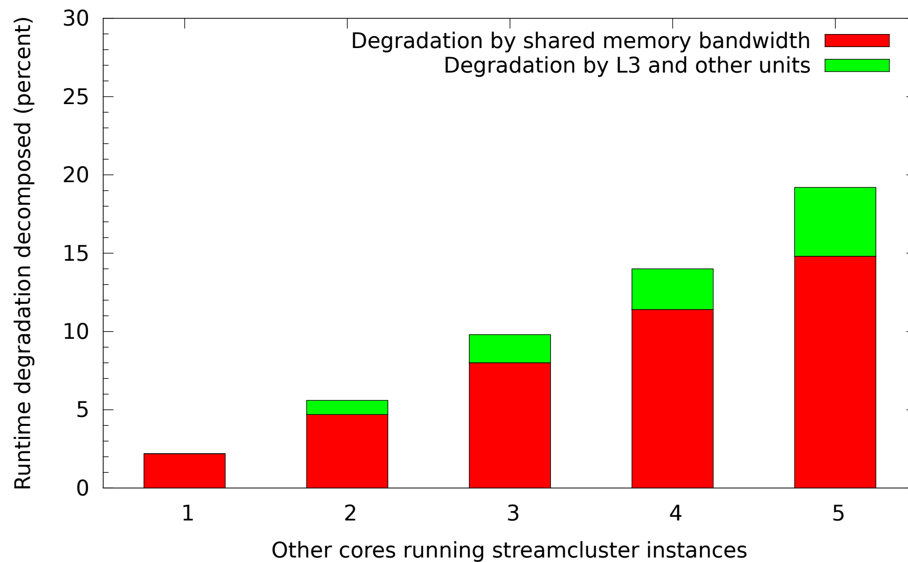


Figure 5.5: Decomposition of runtime degradation by shared memory bandwidth versus L3 and other memory subsystems.

of local memory. This figure shows that the latency that is caused by accessing remote memory does not affect the L3 cache miss rate significantly. Thus, by comparing the slowdown in both scenarios, we can have a rough estimate of what percent of performance degradation is caused by using the memory bandwidth or by the other memory subsystems (most notably L3 and prefetcher). Figure 5.4 shows the runtime degradation comparison of the two experiment setups. Using this information, we can roughly estimate how much runtime degradation is caused by sharing the memory bandwidth.

Figure 5.5 shows the amount of runtime slowdown as we add load generating streamcluster instances and its decomposition into memory bandwidth and other memory subsystems. This experiment shows that *although there is performance degradation when streamcluster instances share the L3 cache and other memory subsystems of their die, it is not significant compared to the performance degradation when the memory bandwidth is shared between all of them.*

$$\text{L3 miss rate} = \frac{\text{L3 cache miss event}}{\text{Retired instructions}} \tag{5.1}$$

Ultimately, the performance degradation caused by sharing L3 depends on the memory access pattern and locality of an application. As a result, we cannot conclude that since streamcluster shows this behavior, all the other applications would show a similar behavior. However, S. Blagodurov et al. [21] performed a similar study and discovered that in most of the current benchmarks, sharing L3 is not the dominant factor in performance degradation. Our evaluation of performance isolation with the shared cache in section 7.3 confirms these findings.

## 5.4 Summary

In this chapter, we discussed the requirements for a multicore operating system to provide performance isolation support. Most of these requirements are easy to provide just by careful placement of processes on cores or memory locations. We studied the case of providing memory bandwidth isolation without hardware support and we came up with a simple estimation to prevent overcommitting the shared bandwidth. We also elaborated on the argument that the degradation caused by sharing the L3 cache is not the most significant factor in total performance degradation of an application.

# Chapter 6

# A resource management subsystem for Barrelfish

In this chapter, we discuss the design and implementation of a resource management subsystem for Barrelfish. We first discuss how we represent the memory subsystem in Barrelfish. Using this representation, we design a resource manager which exports an API which allows resource allocation by taking into account both performance properties and/or performance isolation.

## 6.1 Representation of memory subsystem in Barrelfish

The information regarding the structure of the memory subsystem needs to be represented in a detailed manner for providing a better performance and isolation support. As mentioned before, Barrelfish's approach for representing the diverse set of available hardware is through SKB. We now go over the related schemas for this thesis and how they are populated with facts.

**cpu_thread(APICID, Package_ID, Core_ID, Thread_ID).**

This schema describes the mapping between an execution context APICID to its physical location in the system. The physical location is expressed in the form of Package_ID (the processor ID), Core_ID (the core ID) and Thread_ID (the hardware thread ID). Thread_ID is relevant when cores have hardware threading support. This schema is populated using the *cpuid* instruction on x86 [28, 29].

**cpu_affinity(APICID, _, ProximityDomain).**[1]

This schema describes the mapping between execution context APICIDs to their *ProximityDomain*. ProximityDomain is a unique ID for different NUMA nodes. This information means which core is attached to which memory controller. This schema is populated using the information stored in ACPI tables [30].

**memory_affinity(Base, Limit, ProximityDomain).**

This schema describes the mapping between physical memory locations to ProximityDomains. Physical memory locations are expressed in ranges in the form of (Base, Limit). This information means which physical address ranges belong to which NUMA node. This schema is also populated using the information stored in ACPI tables.

**cache_share(CoreID1, CoreID2, Level).**

If two cores share a level of cache, the information stored here. Unfortunately, there is no unified way of retrieving it. For example, in AMD, if some cores have the same proximity, it is always assumed that the L3 of those cores is shared. Thus, it is difficult not to use some hardcoded facts to get this information. There is a lot of research work on developing methods for automatically calibrating this information (for example, J. Dongarra et al. [38] or K. Yotov et al. [39]). These methods however are out of the scope of this thesis and we will rely on information out of the CPUID instruction and some hardcoded facts. In the current implementation, we do not rely on the information in this schema.

**node_distance(ProximityDomain1, ProximityDomain2, Distance).***[2]

This schema describes the NUMA link distance between two NUMA domains. This information can be found in the SLIT ACPI table. The schema is populated during system startup.

**hypertransport_link(From, To, LinkID).***

This schema contains the interconnect routing table for AMD systems. This information is in the PCI configuration space. Using the routing table and the distance

[1] Underline (_) indicates that the argument is not relevant for this thesis.

[2] Schemas which end with * are created by the author for the purpose of this thesis.

information, it is possible to figure out the interconnect topology. An algorithm is provided later in this section for this exact purpose.

**mem_latency(CoreID, ProximityDomain, Latency).***

The memory access latency for each core to each NUMA node is stored in this schema. The information in this schema and the next one is gathered using microbenchmarks which we will describe later in section 6.2.1.

**mem_bandwidth_read(CoreID, ProximityDomain, Bandwidth).***
**mem_bandwidth_write(CoreID, ProximityDomain, Bandwidth).***

The maximum available read or write memory bandwidth of each NUMA node to each core is stored in this schema. The schema is populated using a microbenchmark during system startup.

## 6.2 A resource management subsystem for Barrelfish

We describe the building blocks of our resource management subsystem (which we call RCM) and the design decisions we make in this section.

### 6.2.1 Microbenchmarks to detect memory performance properties

To provide resource allocation with desired performance properties, we need to use small microbenchmarks to detect these properties. These microbenchmarks populate `mem_latency`, `mem_bandwidth_read` and `mem_bandwidth_write` SKB schemas. The bandwidth properties are also used to implement `get_maximum_bandwidth` routine used in the estimation we described in section 5.2.1.

We use the ideas from the Corey OS Linux benchmark [40] to measure memory latency and bandwidth. To measure the read bandwidth, we use a sequence of load instructions each of which accessing a cacheline of a page [3] in increasing order. We measure the time to read a certain number of pages using this sequence of instructions. The read bandwidth equals:

[3]Page is the smallest granularity in which operating system manages memory. In Barrelfish and on the x86 machines, it is 4KB.

$$\text{Read bandwidth (MB/s)} = \frac{\text{Number of pages} \times \text{Page size}}{\text{Time in second} \times (1024 \times 1024)} \tag{6.1}$$

By changing the core which executes these instructions and the location where the pages are allocated from, we measure the read bandwidth of each core to each NUMA domain. For write bandwidth, we use the store instructions instead of load instructions.

Measuring memory latency is a bit tricky. We need to make sure that each instruction pulls in a cacheline. For this to happen, the cache should not contain the data we are accessing and the prefetcher should not prefetch any data that we are going to access. To make this possible, we first read certain number of bytes from memory to make sure that the cache does not contain the data of the experiment, and then we access a set of cachelines in a random order. Since a prefetcher might prefetch the next and previous cacheline, we always make sure that the next cacheline that we are going to access is not an immediate neighboring cacheline. Similar to the bandwidth microbenchmark, we repeat the microbenchmark for each core and NUMA memory combination. The memory latency is:

$$\text{Memory latency (cycles)} = \frac{\text{Cycles to access all cachelines}}{\text{Number of cachelines accessed}} \tag{6.2}$$

The interested reader can read the source code of the Corey Linux benchmark for more detailed information.

One interesting observation with the Corey Linux benchmark on our Magny-Cours machine when running Linux was the asymmetry we observed in local memory latency of different NUMA nodes. We measured the latency difference of 21% between the node with the best latency and the node with the worst latency. This is most likely because of the physical locations of processor packages with respect to the physical locations of memory banks in the system.

Runtime duration of these microbenchmarks depends on the number of cores and NUMA nodes on the system. On our 4P Magny-Cours machine the microbenchmarks take more than ten minutes to finish. As a result, it is not feasible to run them each time the operating system boots. Instead, after running them once, their results are stored and reused each time the system boots.

### 6.2.2 Monitoring memory bandwidth consumption

We need to monitor the memory bandwidth consumption of each core to make sure that the shared memory bandwidth is not overcommited. These statistics are helpful

for implementing `get_current_bandwidth` and `core_max_consumption` routines used in the estimation in section 5.2.1 to decide whether another process can use this memory bandwidth.

To do this, RCM creates a thread on each core in the system. These threads update the current memory bandwidth consumption by using the performance monitoring counters. To make minimal interference, after reading the counters and updating memory bandwidth consumption, the threads keep dispatching the next process until the next update time. Currently, RCM threads update the memory bandwidth consumption every second. We calculate the read and write bandwidth consumption using the following formulas [27]:

$$\text{Read bandwidth (MB/s)} = \frac{\text{System read events} \times 64}{\text{Time in seconds} \times (1024 \times 1024)} \tag{6.3}$$

$$\text{Write bandwidth (MB/s)} = \frac{\text{System write events} \times 16}{\text{Time in seconds} \times (1024 \times 1024)} \tag{6.4}$$

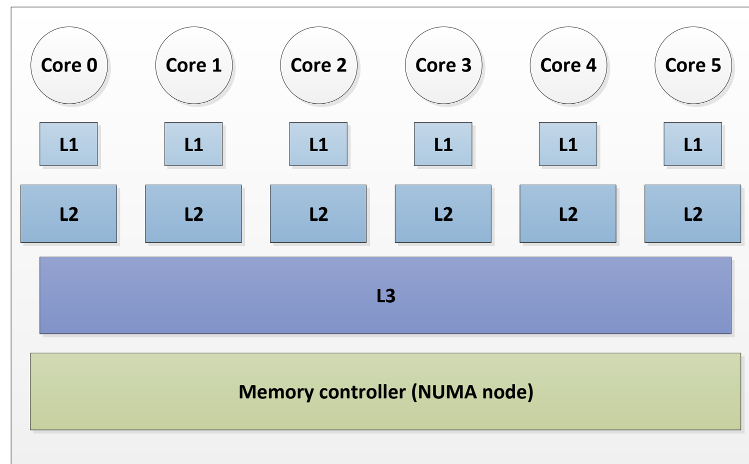


FIGURE 6.1: The topology representation of one the of Magny-Cours dies in RCM.

### 6.2.3 System topology using the information stored in SKB

To provide support for resource allocation and performance isolation, RCM needs to know the exact representation of the system topology. By system topology we mean the mappings of hardware threads to cores, cores to caches and caches to memory controllers. Figure 6.1 shows an example of a system topology. To create the system topology, RCM uses `cpu_affinity`, `memory_affinity`. Currently, we assume that each NUMA node has a shared L3. Extending the system to detect cache sharing at other cache levels

based on the information in SKB is left for future work. We also suppose that SMT is turned off.

RCM stores the performance information for each core using the `mem_latency` and `mem_bandwidth` SKB schemas.

### 6.2.4 An algorithm to create the interconnect topology

To support interconnect-aware performance isolation, RCM needs to know the interconnect topology. The interconnect routing table is available in `hypertransport_link` SKB schema. However, this information is not enough for the purpose of performance isolation. when NUMA nodes are connected using more than one interconnect link, one intermediate node is used to route the data. The routing table reports that a certain link is used to transfer data or coherency message to two or more different nodes and it is not clear which node will do the routing. Using NUMA distance table stored in `numa_distance` SKB schema, we can learn about the routing nodes; the node with distance one will be the immediate routing node when a node uses the same link to connect to different nodes.

We designed the following algorithm to calculate the interconnect path between two NUMA nodes in RCM:

```
1  Interconnect_path(NUMA src, NUMA dst)
2
3      List links = empty_list;
4
5      Distance D = get_numa_distance(src, dst);
6      while D != 0
7          for all links L in src:
8              if L connects src to dst
9                  add L to links
10         for all numa nodes n:
11             if L connect src to n
12                 if get_numa_distance(src, n) == 1
13                     src = n
14                     D--
```

Listing 6.1: Interconnect path discovery algorithm

The algorithm relies on the fact that if an interconnect link is used to connect a source node to a destination node, then there always exists a node which is directly

connected to the source node using that link (line twelve). Since this assumption always holds, the algorithm always terminates with the connecting interconnect links stored in `links` (line nine).

### 6.2.5 Resource management using Barrelfish capability system

To ensure that the isolation guarantees provided by RCM always hold, we need to make sure that there is no possible way for a process to use cycles of another process' isolated core or memory bandwidth. For this purpose, we use Barrelfish's capability system which is based on that of seL4 [41]. The capability model of seL4 makes sure that capabilities cannot be forged or be modified without permission and that is exactly what we are interested in.

We define two new capability types for resource management: RCMCore and RCM-Mem. RCMCore is a capability with an ID of the core[4] which the holder of the capability can execute on and RCMMem holds a NUMA physical memory range which the owner of this capability can allocate memory from. We need to modify a number of Barrelfish's internal subsystems to require these capabilities:

- The spawn daemon. Spawnd has an instance on each core in the system and it is responsible for spawning new processes in the system. We modify this subsystem to require a RCMCore capability and a RCMMem capability. We make sure that the process has the privilege of executing on this core by checking its RCMCore capability. The spawn daemon also loads the binary to the location indicated by RCMMem and passes this capability to the process.
- The memory server. This server is responsible for memory allocation to different processes in the system. With our modifications, the memory server requires a RCMMem capability and it will allocate the memory from the memory range of that capability.
- The monitor. Monitor is a special user mode application which can perform a number of privileged tasks. One of these tasks is capability creation. We add the functionality to monitor for creating RCMCore and RCMMem capabilities for RCM. Monitor also has an instance on each core in the system.

We divide the system into two zones, isolation zone and normal zone. The cores in the normal zone do not provide any kind of isolation grantees (i.e. applications do not need any capability to execute on them). When an application wishes to execute

[4]This is a unique integer which Barrelfish assigns to each online core in the system.

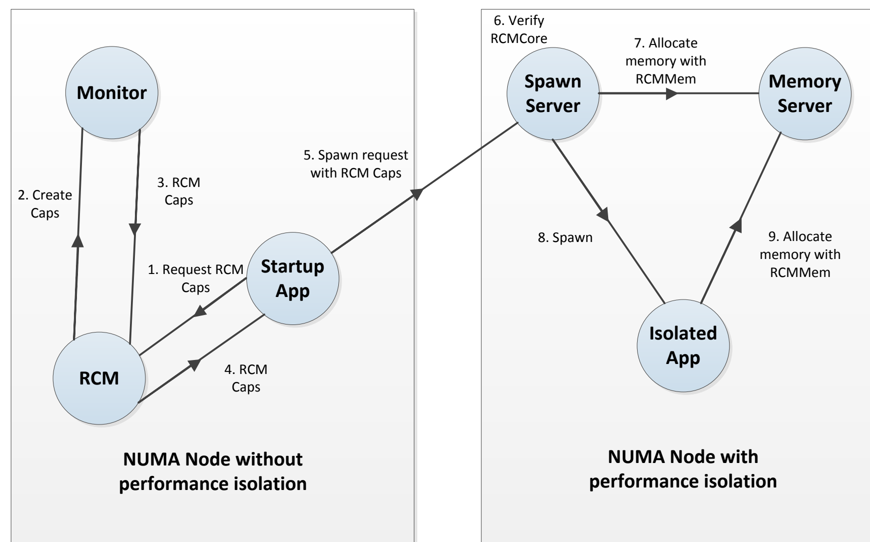


FIGURE 6.2: Steps for executing an application in Barrelfish with the new subsystems.

in the system, it first needs to acquire a RCMCore and a RCMMem capability from RCM. For this purpose, the application can run a small code in the normal zone to acquire a RCMCore and a RCMMem capability with requested isolation properties for the isolation zone. It can then use these capabilities to run under performance isolation. Figure 6.2 shows the the steps required for an application to execute in a node with performance isolation support.

We describe a library in section 6.2.7 which simplifies executing a new application under performance isolation.

### 6.2.6 An API for multicore resource management

In this section, we talk about an API for allocating system resources from RCM. RCM exports this API and applications can allocate resources with different performance or isolation guarantees using it. This API invocation takes place at the step one of figure 6.2.

The set of performance properties which applications can specify are:

1. Memory latency
2. Memory read bandwidth
3. Memory write bandwidth

RCM returns with best available core to memory allocation according to one of the specified desired performance properties. RCM makes this decision based on the result of the microbenchmarks we discussed in section 6.2.1.

The isolation properties which applications can specify are:

1. Core to memory isolation: the core and the path way to memory (i.e. interconnect links) needs to be isolated.
2. Memory controller: the shared memory bandwidth will be isolated and RCM controls further allocations to make sure that the memory bandwidth is not overcommitted.
3. Cache: the application is requesting to run on an isolated cache.

Any of the above performance isolation properties may be specified. RCM's API supports all the combinations of any of the isolation properties with one of the performance properties. Additionally the API also supports:

1. NUMA-aware allocation with a specified performance property.
2. Core allocation on an isolated cache upon presenting the RCMCore capability which owns the cache.
3. Core and memory allocations without isolation guarantees.

To answer all these allocation requests, the system topology described in section 6.2.4. is walked and upon finding a matching candidate, the information regarding its allocation and/or isolation properties is updated. A more detailed information regarding this API is in appendix B.

### 6.2.7 A library for interacting with RCM

As we see in figure 6.2, executing an application in the new system seems rather complicated. To address this, we created a library which interacts with RCM. The library takes care of step one, four and five by implementing the functionality for communication with RCM and the Spawn server. The startup part is a small code (Startup App in the figure) which links against this library and executes the main application by calling a desired function in the library with the path to application and its arguments plus desired performance or isolation properties.

For convenience, the Barrelfish's shell is also using this library and can act as a startup code to execute the main application in isolation. An example of using Barrelfish's shell to spawn an application under performance isolation is in appendix C.

### 6.2.8 Limitations of RCM

We elaborate on a number of limitations in the current design of RCM.

**Memory sharing** In the current implementation of Barrelfish, processes can easily share memory by copying their memory capability to a specified destination. This memory sharing utilizes the bandwidth of the interconnect path connecting these two nodes. We violate one isolation property, if this path would be isolated at any point during the execution of these two processes.

Solving this problem is not trivial since copying capabilities is used as a functionality in Barrelfish and there is no easy way to control capability copying.

Another problem can happen inside an application: Since an application can ask for multiple isolated core and memory, it will have access to different RCMCore and RCMMem capabilities and it can use every combination of the two it desires and this can violate interconnect links isolation like above. There are some possible solutions to this problem:

1. The memory server also asks for the RCMCore capability and verifies with RCM whether the path is isolated for this specific core and memory.
2. RCMCore and RCMMem merge into a single capability. While this has its advantages, it also has limitations. For example, it is not trivial to ask for more memory unless you provide a core for it. This is because RCM needs to know about which cores access which memory locations. It also limits the flexibility of the application to manage its allocated resources.
3. Introducing another capability for interconnect links (e.g. RCMLink) and providing them to the memory server for verification. The memory server needs to have the interconnect topology as well.
4. Applications need to pass on their unique ID to RCM and RCM isolates all the possible core to memory paths of this application. The problem with this approach is under utilization since RCM implicitly isolates interconnect links which the application might never use.

**Lack of process migration** There is no process migration functionality in Barrelfish and this has a number of indications in the current design. The processes that require performance isolation should execute first since as applications start executing and spreading over the isolation zone, it will become unlikely to answer the requests for an isolated core or cache.

In a system with process migration, a resource manager can move processes around the system to answer some performance isolation requests which otherwise was not possible.

This also makes it difficult to measure the effectiveness of the performance isolation provided in the current version of RCM since the mapping of processes without isolation guarantees to possible isolated resources like memory bandwidth is static. We describe how we tackle this problem in section 7.3.

**Multiple allocation at once** The current API does not support allocating multiple resources at once. For example, an application cannot ask for six isolated cores at once and instead it needs to do it iteratively. This can lead to classical deadlock scenarios where applications hold resources and waiting for other applications to release more resources.

**Cache sharing** In the current version of RCM, we assume that each NUMA node has a separate L3 and it is shared between the cores of that NUMA node. While this is true for the systems we are working with in this thesis, this assumption is generally false. There is a need for a library to detect cache sharing information at different levels (L2, L3 and possibly L4 in the future) for different architectures and store them in SKB. This is difficult since each vendor has a different way of producing this information and it also tends to change even for the architectures of the same vendor as they evolve.

**Assumption on the availability of local memory** RCM assumes that there is always memory attached to the local memory controller. It is certainly not the case in most of the systems or with systems that support memory hot plugging.

## 6.3 Summary

We discussed in detail the design and implementation of a resource management subsystem for Barrelfish which support resource allocation with desired performance properties or performance isolation. The current implementation support resource allocation

through an API that allows specifying isolation properties with a desired performance property.

To make this possible we used a number of building blocks like online performance measurement microbenchmarks, detecting system memory topology and interconnect topology using SKB, Barrelfish capability system and online measurement of memory bandwidth consumption.

# Chapter 7

# Evaluation

Barrelfish's resource management subsystem (RCM) enables applications to express two major properties of the resources they are interested in:

1. Performance properties: the best path from an arbitrary core to an arbitrary memory location which has certain performance properties. The performance properties are expressed in terms of lowest latency or highest read or write bandwidth.

2. Isolation properties: the resources that RCM allocates should be isolated from the effects of other applications in the system. The isolation properties are expressed in terms of isolated core, isolated memory controller, isolated shared cache and isolated interconnect links.

To evaluate the benefits of RCM when providing these two properties, we have ported four benchmarks from the PARSEC benchmark suite [37, 42] to Barrelfish. We first look at the properties of these benchmarks in section 7.1 and then we evaluate the benefits of RCM with these macrobenchmarks when using performance properties in section 7.2 and when using isolation properties in section 7.3.

We run the benchmarks in single threaded mode with the provided *small* workload from the suite on our 4P Magny-Cours machine which we previously looked at its architecture in section 3.1.1.

## 7.1 PARSEC benchmark suite[1]

We closely look at the benchmarks that we are going to use.

[1] The description of the benchmarks are taken from [42].

**Blackscholes** Blackscholes calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE) [43]. The input is a set of options of the portfolio and the output is the prices for all options which are written to a file.

**Canneal** This kernel tries to find the minimum routing cost for designing chips using cache-aware simulated annealing [44]. The input to Canneal is a file which contains netlist data structure elements, and the configuration inputs to the simulated annealing algorithm. It outputs the cost of final routing to the console.

**Fluidanimate** This Intel RMS application uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes [45]. The input to fluidanimate is a file containing a number of particles which describe the fluid. The algorithm output which is written to a file can be visualized by detecting and rendering the surface of the fluid.

**Streamcluster** Streamcluster solves the online clustering problem [46]: For a stream of input points, it finds a predetermined number of medians so that each point is assigned to its nearest center. The quality of the computation is calculated using sum of squared distances of the points to the center of their cluster. The input to the Streamcluster is a set of configuration options which describe the random dataset to be generated and the output is cluster information which is written to a file.

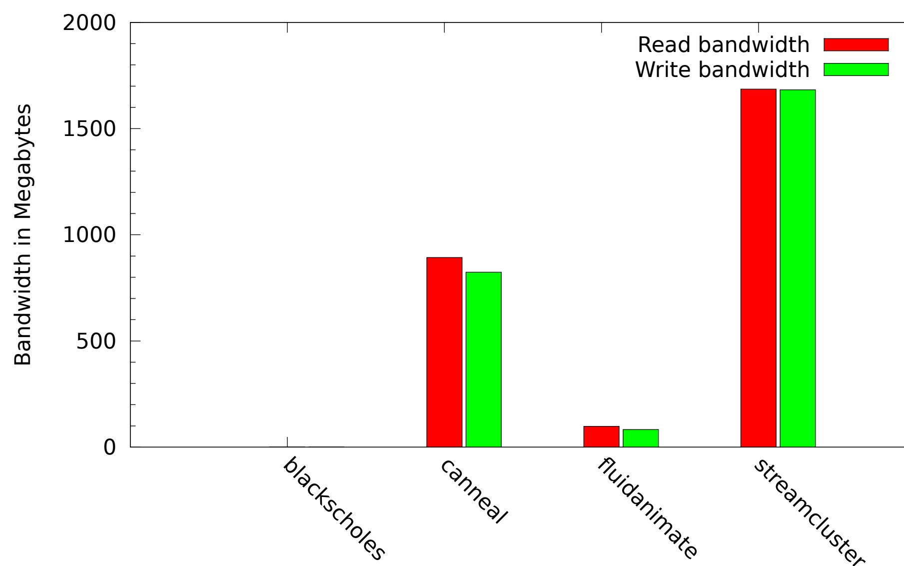


FIGURE 7.1: The memory bandwidth usage of the four ported benchmarks from PARSEC.

Now we look at the properties of these benchmarks which are of interest for this thesis. Blackscholes, Canneal and Fluidanimate read their input from a file and Blackscholes, Fluidanimate and Streamcluster write their output to a file as well. Since we do not address I/O in this thesis, we only measure the performance data for the execution period and we discard the initialization phase where the data is loaded from file to memory and the part where the results are written back to a file. We call the execution period the execution phase from now on.

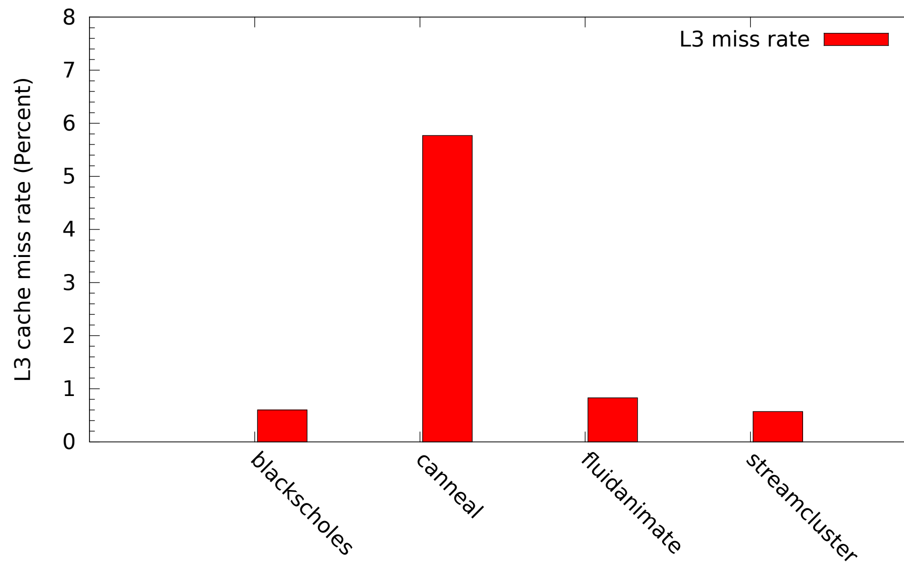


FIGURE 7.2: The L3 cache miss rate of the four ported benchmarks from PARSEC.

Figure 7.1 shows the measured memory bandwidth of the chosen benchmarks and figure 7.2 shows the L3 cache miss rate. Canneal and Streamcluster have relatively higher memory bandwidth usage and are the most interesting ones in terms of provided shared memory bandwidth. Blackscholes' dataset fits into the L2 cache and as a result it does not consume considerable memory bandwidth during its execution phase. Canneal is more cache unfriendly compared with the other three benchmarks since it has the highest cache miss rate.

## 7.2 Benchmarking performance properties

We show that the many possibilities of process placement and the location of memory result in variation in total execution time because of different memory latency or bandwidth. There are even asymmetries in observed memory latency when cores allocate memory locally as we discussed in section 6.2.1. We show that by using RCM, applications can have ideal execution time by expressing the memory performance property that they are interested in.

For each of the benchmarks, we measure their runtime on different NUMA cores accessing different NUMA domains. Since the cores of each NUMA domain have similar performance when accessing memory, we take one core from each NUMA domain. Since the Magny-Cours machine has eight NUMA domains, we run 64 experiments for each of the benchmarks. From these 64 runtimes, we extract the best and worst runtimes and compare them with a runtime when the benchmark runs using RCM. We choose latency as our desired performance property since it is clear that the consumed bandwidth of the benchmarks are considerably less than the available main memory bandwidth on each NUMA node.

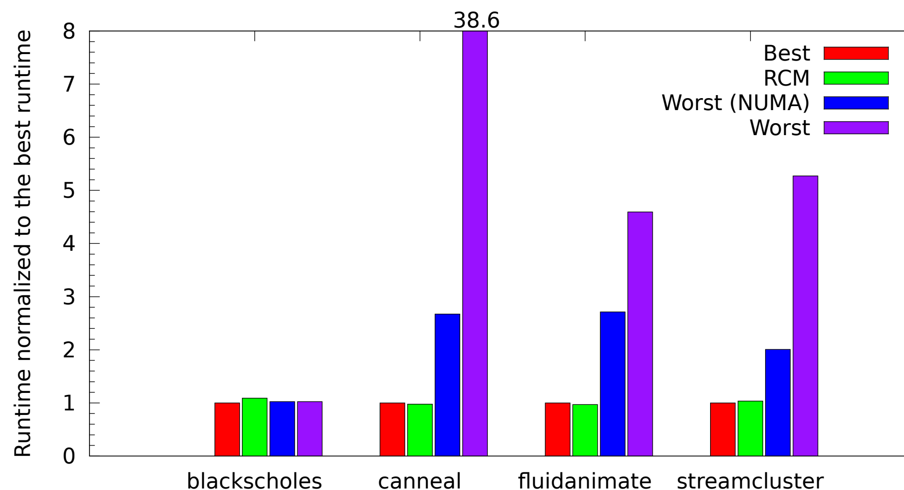


FIGURE 7.3: Runtime comparison of the PARSEC benchmarks. Best and worst runtime are obtained by trying all possible core/memory allocation. RCM is when the core and memory are allocated by RCM.

The normalized results with the best runtime is in figure 7.3. Except Blackscholes which its workload fits into the L3 cache, all other benchmarks show a considerable difference in their execution time (Look at `Best` and `Worst`) with regard to their core and memory placement. Canneal shows off-the-chart difference since it has a high cache miss rate and as a result high latency of accessing remote memory degrades its execution time significantly. Due to asymmetries in observed local memory performance in different NUMA nodes, NUMA-aware allocation can have suboptimal performance (look at `Worst (NUMA)`). RCM always allocates the optimal core and memory due to its awareness about latency of all cores to memory locations. With Canneal and Fluidanimate RCM performs about 3% better than the best measured allocation with the default Barrelfish process allocator. We believe this is because RCM enforces NUMA-aware loading of binary image while it is not the case in the default implementation. We however did not verify this due to time constraints.

This experiment shows that applications can benefit from performance-aware resource allocation inside multicore operating systems.

## 7.3 Benchmarking isolation properties

A performance isolation benchmark should be able to show two desirable properties of the system:

1. If an application is running with performance isolation guarantees then its performance should not decrease if the system is overloaded with other applications.
2. If an application is running with performance isolation guarantees, the performance of the other applications in the system should not dramatically decrease.

RCM provides isolation at:

1. Core by not placing any other process on an isolated core.
2. Interconnect by isolating interconnect links.
3. Cache level by not placing any other process on the cores which share a cache with an isolated cache.
4. Shared memory bandwidth by making sure that the shared memory bandwidth is not overcommitted.

The benefits of running one application on a core is obvious and indications of it on other running applications fall into the area of scheduling and we do not address it here. Isolation of interconnect links becomes interesting when interacting with I/O devices. Since we mostly allocate memory locally, we leave it for future work. RCM however is interconnect aware and isolates interconnect links whenever necessary during our experiments. We study the more interesting cases of isolated cache and isolated memory bandwidth.

Due to one of our design choices which we discussed in section 5.2.1, we cannot migrate processes. It means that when it is detected that the shared bandwidth is overcommitted, it is not possible to migrate some of the processes that are running in this NUMA domain to another NUMA domain to ensure that performance isolation guarantees are not violated. This limitation affects the way we can design our performance isolation benchmark. This is because we do not have the knowledge of memory

bandwidth requirements in advance. To cope with this limitation, we take an iterative approach when running the instances of the benchmark. We explain our approach shortly.

As discussed in section 6.2.5, we divide our Magny-Cours machine into two zones. One with performance isolation support and the other for applications without any performance isolation requirement. The isolation zone consists of four NUMA nodes with 24 cores. We remove Blackscholes from the set of our benchmarks since it has no interesting property which results in resource sharing. We use five instances of Canneal, five instances of Fluidanimate and fifteen instances of Streamcluster to run together in the system. We use more Streamcluster instances due to its high bandwidth consumption which makes it possible to saturate the main memory bandwidth. We thus run 25 instances in the isolated zone in each of the measurements. This number is chosen to make sure that without any isolation guarantee, all the cores run at least one instance.

We first let all the instances load their raw data (Streamcluster does not have this phase) once and then we run their execution phase in an infinite loop. The order in which we run the instances is as follows:

1. We first run one of the three benchmarks in one of the two different isolation guarantees (isolated shared memory bandwidth or isolated cache).
2. When the first ten loops of the execution phase of benchmark finishes we run the next one of the remaining two benchmarks without any isolation guarantee.
3. We iteratively add other instances from the three benchmarks as soon as the first benchmark loop of the last one finishes.
4. As we are finished with fifteen instances of the benchmarks (five instances of each three benchmarks), we iteratively add the rest of Streamcluster instances.

After all the instances are running their execution phase in a loop, we measure the runtime of the execution phase of the instance under performance isolation. We also measure the runtime of each one of the other instances in the system. We repeat the experiment without any application running in performance isolation.

Figure 7.4 shows the normalized results of running the experiment. The red bars are when the applications are running alone in the system and are chosen as the baseline. The green bars are when the isolated benchmark is running with an isolated L3 along with 24 other instances in the system and the purple bars when the isolated benchmarks are running with memory bandwidth isolation with 24 other instances in the system.

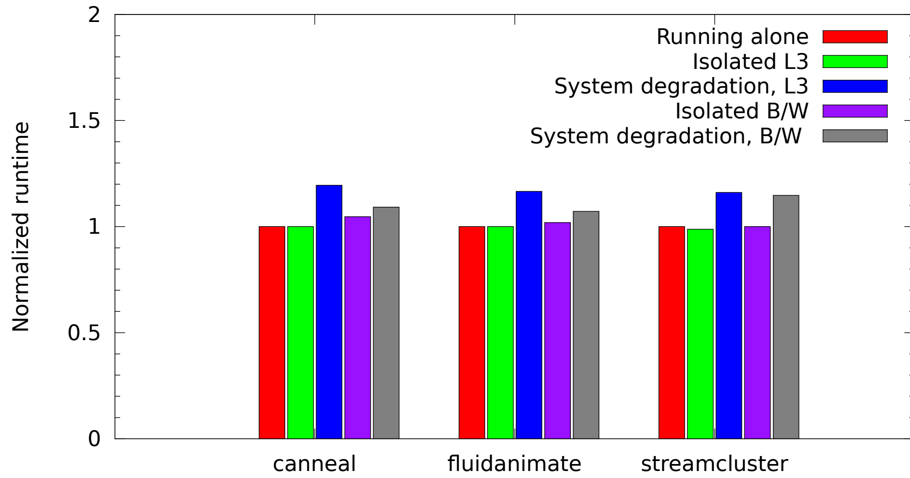


FIGURE 7.4: Runtime degradation of the instance which is running with cache or memory bandwidth isolation and normalized average runtime degradation of all the other instances in the system.

The blue and gray bars are the average runtime degradation of all the instances in both isolation scenarios. We call this metric total system degradation. In this benchmark, total system degradation is the sum of runtime values of all instances when there is one isolated application in the system normalized with when there is no isolated application running in the system. Formally, formulas 7.1, 7.2 and 7.3 are used to calculate the total system degradation:

$$R^{\Diamond} = \sum_{i=1}^{n_B} \sum_{j=1}^{n_{B_i}} R^{\Diamond}_{i,j} \tag{7.1}$$

Where $R^{\Diamond}$ is the sum of the runtimes of all instances in the system, $n_B$ is the number of different benchmarks that are running in the system, $n_{B_i}$ is the number of instances of the $i$th benchmark and $R^{\Diamond}_{i,j}$ is the runtime of instance $j$ of $i$th benchmark, all when there is no isolated instance in the system.

$$R^{\triangle} = \sum_{i=1}^{n_B} \sum_{j=1}^{n_{B_i}} R^{\triangle}_{i,j} \tag{7.2}$$

Where $R^{\triangle}$ is the sum of the runtimes of all instances in the system and $R^{\triangle}_{i,j}$ is the runtime of instance $j$ of $i$th benchmark, all when there exists an isolated instance in the system.

Finally, the total system runtime degradation denoted by $D_{system}$ is:

$$D_{system} = \frac{R^{\triangle}}{R^{\diamond}} \tag{7.3}$$

**Discussion:** With all of the instances we see a similar pattern: when the isolated instance is running with an isolated L3, its runtime duration does not increase at all. However, since by isolating L3 no other application can run on the other cores of the die with isolated L3, the other 24 instances should run on 18 cores instead of 23 available cores. This results in upto 20% of observed system runtime degradation which as described measures the average runtime degradation of all instances in the system. When the isolated instance runs with isolated shared bandwidth, in the worst case its runtime duration increases by 5%. This is expected because of the cache pollution effect. The system runtime degradation decreases by 14%. It is less than the L3 isolation because RCM allows benchmark instances to run in the cores of the die with memory bandwidth isolation as long as memory bandwidth is not overcommitted.

It is clear that there is a trade-off coming with performance isolation. As we under-utilize system resources to provide performance isolation to an instance, the performance of other instances decrease. The finer level of provided performance isolation results in greater under-utilization of the system resources which in turn decreases the overall performance of the system. With RCM, we can decide the level of isolation we are interested in by choosing between running on an isolated cores, cache or memory bandwidth.

Another interesting benchmark is to show the total runtime degradation of all the applications without performance isolation guarantees when the number of applications with isolation guarantees increases. To do this, we repeat the previous benchmark, but this time we isolate more instances. We first isolate Canneal, then we isolate Fluideanimate and finally we isolate Streamcluster. In this experiment, it is only possible to isolate memory bandwidth or L3 of three instances in the system so that other instances can run in the rest of isolation zone.

**Discussion:** Figure 7.5 shows the result. The runtime of isolated instances with L3 isolation do not degrade. We can observe that as we isolate more L3 caches, the system runtime degradation increases dramatically. When there are three isolated L3, 22 instances should run on six cores. As a result, instances are running at about three times slower. With isolated memory bandwidth the average runtime of isolated instances decreases by at most 5%. The system runtime however is much less compared with the L3 isolation. With three isolated instances, there is 47% total system degradation.

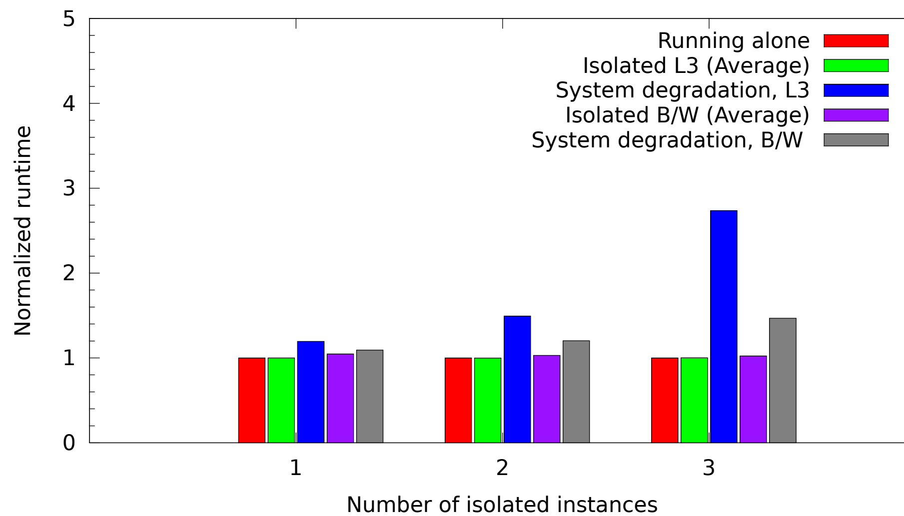


FIGURE 7.5: Performance isolation experiment with multiple isolated instances.

This experiment shows that it is possible for applications with performance isolation support to coexist in a multicore systems. However, it becomes more costly as we increase the number of isolated instances in the system.

## 7.4 Summary

We described four macrobenchmarks that we ported to Barrelfish to show the benefits of our resource management subsystem.

We first showed the benefits of performance-aware resource management and then we showed the effectiveness of RCM in bringing performance isolation by designing a performance isolation benchmark using a mix of the ported benchmarks. We also discussed the trade-offs of running applications under performance isolation in multicore systems.

# Chapter 8

# Conclusion

We looked at the architecture of two modern x86 multicore systems to study how they differ from commodity systems in the way system resources are shared. In these systems, inter-processor main memory bandwidth, different levels of caches and the system interconnect are implicitly shared between cores, but their allocation is not in the direct control of the operating system. Using a synthetic microbenchmark, we showed this implicit sharing of resources can result in undesired performance degradation of performance critical applications.

Based on the obtained results of the microbenchmark, we discussed the possibility of providing performance isolation on modern multicore hardware by controlling these implicitly shared resources. We came up with the following concrete conditions under which performance isolation becomes possible:

- The shared main memory bandwidth should not be overcommitted when a process under performance isolation is using it.
- In cache-coherent systems, the coherency broadcasts result in excessive traffic on the system interconnect. This can become a performance bottleneck on each processor. Systems with mechanisms like cache directory reduce this traffic and avoid the performance bottleneck. We conclude that hardware can provide performance isolation if such a mechanism exists the system.
- The provided bandwidth on the interconnect links is limited and shared between different cores of a processor. The operating system should provide a mechanism to control this shared bandwidth.
- Some applications may require exclusive levels of cache for performance isolation. The operating system needs to provide a mechanism for such a purpose. However,

we argued that the shared last level cache is not usually the dominant factor in performance degradation.

We then described the design and implementation of a resource management subsystem for Barrelfish called RCM. RCM provides mechanisms for performance aware resource allocation and performance isolation by respecting the mentioned conditions. RCM provides these mechanisms using various techniques like interconnect topology discovery, online microbenchmarks to measure different performance properties, online performance monitoring at core level and architectural information like NUMA groups, NUMA physical memory ranges and shared caches.

Finally, we looked at the benefits of having a subsystem like RCM in a multicore operating system. For that end, we ported some benchmarks from PARSEC to Barrelfish. Using them, we showed that applications can enjoy increased performance by performance aware resource allocation in modern multicore systems. We also showed that performance isolation is possible by the mechanisms that RCM provides and discussed the performance trade-offs of running applications under different degrees of performance isolation.

## 8.1 Future work

We describe possible directions for future work on this subject:

1. Studying the benefits of performance aware allocation and performance isolation for multithreaded applications and I/O workloads.
2. Development of a cache topology discovery which interacts with SKB. This is becoming more important due the diverse set of multicore processors, each having its own architecture-specific way of retrieving this information.
3. Providing a time-slice during which performance isolation is guaranteed to avoid underutilizing the system resources during the complete execution duration of an isolated application.
4. Adding the functionality for applications to specify the amount of guaranteed memory bandwidth they are interested in. RCM can then make a more aware decision for core and memory placement based on the current consumption of main memory bandwidth across all NUMA nodes in the system.
5. Support for process migration in Barrelfish to support *dynamic* mapping of isolated applications to cores and memory regions.

6. A more sophisticated estimation of memory bandwidth consumption of a new application.

7. Complete implementation of RCM's API. For example, currently there is no implementation backend for releasing isolated resources.

# Appendix A

# Terminology

Currently, in the multicore research community, there are many terms used to describe different parts of multicore systems. Some of these terms could mean the same thing. Here, we have come up with a list of these keywords and what we mean by them in this thesis. The terms that are related or used interchangeably are put together.

**Core = physical core:** An abstraction for a unit which has computing capabilities. We base most of the definitions using core.

**Hyperthreading = simultaneous multithreading = hardware threading:** Each core can have multiple execution threads running on it. Operating system usually sees each hardware thread as a core for itself. Thus, if a core has for example four hardware threads, it means that the operating system sees four cores instead of one.

**Virtual core = hardware execution context:** A virtual core is either backed by a hardware thread or a physical core in systems without hardware threading support.

**Die, node, NUMA node:** Die is a small block of semiconducting material on which different building blocks of a processor are implemented. Dies could host one or more than one cores. If a die or a set of dies share the same path to memory, they are called a node. Thus, if a die has cores with a memory controller, it is called a node or NUMA node.

**Processor = package = processor package, multi-chip module(MCM), multi-node processor:** A processor is a die or a set of dies with other units which provide

other functionality required. When a processor is consisted of more than one die, it is called a multi-chip module or MCM. A MCM with more than one memory controller is called a multi-node processor.

**Multicore processor = chip multicore processor = manycore processor:** A processor which has more than one physical core.

**Cache directory, HT assist, probe filter, snoop filter:** Different technologies to reduce interconnect traffic caused by cache coherency protocol messages sent around to implement the memory consistency model of the system.

# Appendix B

# RCM's API

Barrelfish uses an interface description language called flounder [47] to generate messaging or RPC stubs. We use flounder to describe RCM's API. Below is a rather descriptive interface file of RCM which we described in chapter 6:

```
interface rcm "Resource Manager Interface" {

        alias numaid uint8;

        typedef enum {
            MEMORY_LATENCY_SENSITIVE,
            MEMORY_READ_BANDWIDTH_SENSITIVE,
            MEMORY_WRITE_BANDWIDTH_SENSITIVE,
            NO_PREFERENCE
        } MEMORY_PERFORMANCE_INFO;

        /* All the requests are in the form of:
         * rcm_CORE_MEMORY_request
         * cored means we do not care about the location
         * of the core and and memd means we do not care
         * about the location of the mem.
         */

        /* We do not care where the core or memory
         * is coming from
         */
        rpc rcm_cored_memd_request(in bool exclusive_cache,
            in bool core_memory_isolated,
            in bool memory_controller_isolated,
            in MEMORY_PERFORMANCE_INFO memory_performance,
```

```
26              out errval err,
27              out cap core,
28              out cap mem);
29          /* We do not care where the core is coming from
30           * but we care where the memory is coming from.
31           */
32          rpc rcm_cored_numa_request(in bool exclusive_cache,
33              in bool core_memory_isolated,
34              in bool memory_controller_isolated,
35              in MEMORY_PERFORMANCE_INFO memory_performance,
36              in numaid numa,
37              out errval err,
38              out cap core,
39              out cap mem);
40          rpc rcm_cored_local_request(in bool exclusive_cache,
41              in bool core_memory_isolated,
42              in bool memory_controller_isolated,
43              in MEMORY_PERFORMANCE_INFO memory_performance,
44              out errval err,
45              out cap core,
46              out cap mem);
47
48          /* Request for an isolated cache would inevitably
49           * results in isolated core as well.
50           */
51
52          /* We do not care where the memory is coming from */
53          rpc rcm_coreoncache_memd_request(
54              in bool memory_controller_isolated,
55              in cap core_with_cache,
56              in MEMORY_PERFORMANCE_INFO memory_performance,
57              out errval err,
58              out cap core,
59              out cap mem);
60          /* We care where the memory is coming from */
61          rpc rcm_coreoncache_numa_request(
62              in bool memory_controller_isolated,
63              in cap core_with_cache,
64              in MEMORY_PERFORMANCE_INFO memory_performance,
65              in numaid numa,
66              out errval err,
67              out cap core,
68              out cap mem);
69
```

```
70          /* Memory requests with isolation */
71
72          /* We do not care where the memory is coming from */
73          rpc rcm_coregiven_memd_request(in bool memory_isolated,
74              in bool memory_controller_isolated,
75              in cap core,
76              in MEMORY_PERFORMANCE_INFO memory_performance,
77              out errval err,
78              out cap mem);
79          /* We care where the memory is coming from */
80          rpc rcm_coregiven_numa_request(in bool memory_isolated,
81              in bool memory_controller_isolated,
82              in cap core,
83              in numaid numa,
84              out errval err,
85              out cap mem);
86
87
88          /* Memory controller isolation */
89
90          rpc rcm_memory_controller_isolation_request(in cap mem,
91              out errval err,
92              out cap mem_isolated);
93
94          /* Resource release */
95
96          rpc rcm_core_memd_release(in cap core,
97              out errval err);
98          rpc rcm_cored_mem_release(in cap mem,
99              out errval err);
100         rpc rcm_core_mem_release(in cap core,
101             in cap mem,
102             out errval err);
103         rpc rcm_memory_controller_release(in cap mem_isolated,
104             out errval err,
105             out cap mem);
106 };
```

Listing B.1: RCM Interface file

Out of these functions, we only have implemented `rcm_cored_memd_request`, `rcm_cored_numa_request` and `rcm_cored_local_request` due to time constraints. In all of the implemented functions, it is possible to ask for an isolated core, isolated cache

and isolated memory bandwidth with a desired performance propertie. Implementation of the rest is left as future work.

# Appendix C

# Using RCM within fish

Barrelfish has an interactive shell called *fish*. We have added to fish the functionality to use RCM for providing application execution under performance isolation. The command is called *rcm*. If we invoke rcm in fish, we see the following:

```
> rcm
Usage: rcm is_isolated(0/1) exclusive_cache(0/1)
memory_load_control(0/1)
memory_performance(0=latency/1=read_bw/2=write_bw)
application [args]
>
```

Listing C.1: Invocation of rcm in Fish

We can execute any application under RCM by defining whether it wants to run on an isolated core (is_isolated), isolated cache (exclusive_cache) or isolated memory bandwidth (memory_load_control) with one desired performance propertie (memory_performance). For example, `rcm 1 1 1 0 test` runs the application `test` with all the performance isolation guarantees with the best available core to memory latency.

# Bibliography

[1] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In Proceedings of the 22nd ACM Symposium on OS Principles (SOSP), 2009.

[2] R. P. Draves, M. B. Jones, P. J. Leach, and J. S. Barrera. Modular real-time resource management in the Rialto operating system. In Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS), 1995.

[3] F. Bellosa. Process Cruise Control: Throttling Memory Access in a Soft Real-Time Environment. University of Erlangen, Technical Report TR-I4-97-02, July 1997

[4] A. Marchand, P. Balbastre, I. Ripoll, and A. Crespo. Providing Memory QoS Guarantees for Real-Time Applications. Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2008.

[5] G. Koren and D. Shasha. Skip-over algorithms and complexity for overloaded systems that allow skips. In Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS), 1995.

[6] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In the Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), 1995.

[7] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. IEEE Journal on Selected Areas in Communications, Volume: 14 Issue:7, 1996.

[8] S. Hand. Self-paging in the Nemesis operating system. In Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI), 1999.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer†, I. Pratt, A. Warfield. Xen and the art of virtualization. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP), 2003.

[10] R. Liu, K. Klues, S. Bird, S. Hofmeyr†, K. Asanovi, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client os. In Proceedings of the Workshop on Hot Topics in Parallelism (HotPar) Usenix, 2009.

[11] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In proceeding of the ACM symposium on Operating Systems Design and Implementation (OSDI), 1999.

[12] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. IEEE/ACM International Symposium on Microarchitectures, 2006.

[13] B. Ahsan and M. Zahran. Managing Off-Chip Bandwidth: A Case for Bandwidth-Friendly Replacement Policy. Workshop on Managed Many-Core Systems (MMCS), 2009.

[14] R. Lee and X. Zhang. MCCDB: minimizing cache conflicts in multi-core processors for databases. In proceeding of 35th international conference on Very Large DataBases (VLDB), 2009.

[15] J. Lira, C. Molina, and A. Gonzalez. Analysis of Non-Uniform Cache Architecture Policies for Chip-Multiprocessors Using the Parsec Benchmark Suite. Workshop on Managed Many-Core Systems (MMCS), 2009.

[16] A. G. Ailamaki, D. J. Dewitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go. In proceeding of 35th international conference on Very Large DataBases (VLDB), 1995.

[17] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, A. Gupta. The impact of architectural trends on operating system performance. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), 1995.

[18] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? Technical Report, Technote 11, Digital Equipment Corporation Western Research Laboratory, 1989.

[19] I. Tuduce, Z. Majo, A. Gauch, B. Chen, and T. R. Gross. Asymmetries in Multi-Core Systems Or Why We Need Better Performance Measurement Units. The Exascale Evaluation and Research Techniques Workshop (EXERT), 2010.

[20] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/ Accessed on Feb. 2011.

[21] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention Aware Scheduling on Multicore Systems, in ACM Transactions on Computer Systems, vol. 28, issue 4, 2010.

[22] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2009.

[23] I. Kuz, Z. Anderson, P. Shind, and T. Roscoe. Multicore OS benchmarks: we can do better. 13th Workshop on Hot Topics in Operating Systems, 2011.

[24] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In Proceedings of the Workshop on Managed Many-Core Systems (MMCS), 2008.

[25] The ECLiPSe Constraint Programming System, http://www.eclipseclp.org/ Accessed on Feb. 2011.

[26] P. Mochel, The sysfs Filesystem, Proceedings of Annual Linux Symposium, 2005.

[27] P. J. Drongowski, Basic Performance Measurements for AMD Athlon$^{TM}$ 64, AMD Opteron$^{TM}$ and AMD Phenom$^{TM}$ Processors, Advanced Micro Devices, Inc., Boston Design Center, 2008.

[28] Intel Processor Identification and the CPUID Instruction, Application Note 485, Jan. 2011.

[29] AMD CPUID specification, Publication 25481, Revision 2.34, Sep. 2010.

[30] Advanced Configuration and Power Interface Specification, Revision 3.0a, Dec. 2005.

[31] A. Kleen, A NUMA API for Linux, SUSE Labs white paper, Aug. 2004.

[32] Advanced Micro Devices, BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors, Mar. 2008.

[33] Linux Programmer's Manual, Accessed on Mar. 2011.

[34] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Blade Computing with the AMD Opteron$^{TM}$ Processor ("Magny-Cours"). Symposium on High Performance Chips (Hot Chips), 2009.

[35] K. Gharachorloo, MEMORY CONSISTENCY MODELS FOR SHARED-MEMORY MULTIPROCESSORS, Technical Report: CSL-TR-95-685, Stanford university, 1995.

[36] B. Waldecker and P. Conway, AMD Opteron$^{TM}$ Multicore Processors, Joint NERSC/OLCF/NICS Cray XT5 Workshop, 2010.

[37] C. Bienia†, S. Kumar, J. P. Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications, Princeton University Technical Report TR-811-08, 2008.

[38] J. Dongarra, S. Moore, P. Mucci, K. Seymour, and Haihang You. Accurate Cache and TLB Characterization Using Hardware Counters, International Conference on Computational Science (ICCS), 2004.

[39] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters, Proceedings of ACM SIGMETRICS international conference on Measurement and modeling of computer systems, 2005.

[40] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In Proceedings of the ACM Symposium on Operating Systems Design and Implementation (OSDI), 2008. Linux benchmarks used in the paper: `http://pdos.csail.mit.edu/corey/osdi-bench.tar.gz`, accessed on Feb. 2011.

[41] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP), 2009.

[42] C. Bienia, Benchmarking Modern Multiprocessors, Ph.D. Thesis. Princeton University, 2011.

[43] F. Black and M. Scholes. The Pricing of Options and Corporate Liabilities. Journal of Political Economy, 1973.

[44] P. Banerjee. Parallel Algorithms for VLSI Computer-Aided Design. Prentice-Hall, Inc., 1994.

[45] M. Müller, D. Charypar, and M. Gross. Particle-Based Fluid Simulation for Interactive Applications. In Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2003.

[46] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani. High-Performance Clustering of Streams and Large Data Sets. In Proceedings of the 18th International Conference on Data Engineering, 2002.

[47] P. Dagand. Language Support for Reliable Operating Systems. Master's thesis, ENS Cachan-Bretagne – University of Rennes, France, 2009.