CHARACTERIZATION OF CLOUD-BASED APPLICATIONS FOR DIFFERENT MEMORY RESOURCE SHARING SCENARIOS

by

ANA MARIA BERNAL MONTANA, B.Sc.

THESIS
Presented to the Graduate Faculty of
The University of Texas at San Antonio
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

COMMITTEE MEMBERS:
Lide Duan, Ph.D., Chair
Eugene John, Ph.D.
Wonjun Lee, Ph.D.

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Engineering
Department of Electrical and Computer Engineering
May 2017

DEDICATION

This thesis is dedicated to my husband Roger. Thank you for your encouragement, unconditional support and infinite patience during my time at graduate school.

ACKNOWLEDGEMENTS

First, I would like to thank Dr. Lide Duan for giving me the opportunity to work with him, teaching me how to do research, how to think better and more critically. For his guidance, helpfulness and patience throughout the development of this work, I will always be very grateful.

Thanks to Dr. Eugene John, for providing me with a different perspective of the complexity behind the design and fabrication of computers and for his dedication in making sure each one of his students, me included, understand the concepts of VLSI and learn by doing.

Thanks to Dr. Wonjun Lee, because due to his detailed explanations and his industry approach when highlighting the most important aspects of the programming languages learned in his course, I was able to improve significantly my programming skills. This has served and will continue to serve me immensely on my academic and professional life.

Thanks to Dr. Jeff Prevost, for sharing his industry experience on his Cloud Computing course. The concepts I learned in his class were very useful in many challenges I encountered during the experiments I conducted for this thesis.

Last but not least, I would like to thank Dr. Wei-Ming Lin for his support, dedication, concern, guidance, remarkable lectures, tough exams and assignments. It was because of this that my approach to solving problems and learning drastically changed for good, and I discovered that Computer Architecture is my passion in life.

May 2017

CHARACTERIZATION OF CLOUD-BASED APPLICATIONS FOR DIFFERENT MEMORY RESOURCE SHARING SCENARIOS

Ana Bernal Montana, M.S.
The University of Texas at San Antonio, 2017

Supervising Professor: Lide Duan, Ph.D., Chair

A typical Cloud-Based application is scale-out, latency sensitive, and usually involves processing of enormous amounts of data. Web Searches, Streaming services, online transactions, etc., run on massive Warehouse-Scale Computers that are powered by processors designed to run traditional scale-up personal computer applications.

Since Architectural decisions for the processors that run the Internet today were made with a desktop computer in mind, memory resources are often overprovisioned and underutilized. In this work, five different Cloud-Based applications from the CloudSuite benchmark suite are characterized according to their interaction with memory resources and their level of data sharing. Threads of each workload are distributed among nodes to study how different scenarios of Last Level Cache and Memory Controller Channel sharing can affect performance. Furthermore, the impact of co-running applications is also taken into consideration, and a scheduling algorithm intended to run on top of the Operating System scheduler is proposed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE: INTRODUCTION

During the past few years, a number of Cloud services have emerged and become a part of everyday life for most people. Social Networks, Web Searches, Streaming Services, etc., are translated into millions of parallel computations running on a cluster of interconnected servers which constitute the foundation of the Cloud itself.

Cloud applications tend to be scale-out, meaning more servers are added as growth demand increases. Even a small percentage of improvement in performance in a single server processor, can make a huge difference in the overall performance of the Warehouse-Scale Computers that constitute the Cloud.

Cloud servers are built with multiprocessor systems, consisting of a group of cores per socket that often share at least one level of cache and the interconnection to memory. This sharing of resources can result in either benefiting or degrading performance. For instance, threads of an application with high levels of data sharing running on the same node (or in a configuration where memory resources are shared) can benefit from this scenario, since a single copy of the data in the shared cache would be used by all active threads, and fewer memory accesses would be required. Moreover, cache coherence and main memory traffic would be reduced.

Threads of applications with low levels of data sharing on the other hand, will contend for resources (because fetching new data will cause cache evictions), resulting in a degradation in performance if placed in the previous scenario of shared memory resources.

An important aspect of modern server multiprocessors is that they were designed with typical desktop scale-up applications in mind, resulting in architectural inefficiencies [1] for Cloud workloads, which are generally scale-out.

Characterizing the memory resource sharing interference on Cloud-Based applications could provide an insight to future server multiprocessor system design, as memory provisioning decisions could be made based on this results.

Although related work [24] has shown that there is no sizable effect from scheduling threads to cores to share memory resources in one way or another, there is evidence [4] that indicates how thread-to-core mapping can improve or worsen performance, depending on the data-sharing characteristics of the application and the distribution of threads to determine how memory resources are to be shared.

In this work, five of the CloudSuite [1][2] benchmarks were selected to study how scheduling threads of Cloud-Based applications under different Last Level Cache and Memory Controller Channel sharing scenarios can have an impact on the performance of servers. Unlike existing Datacenter benchmarks, CloudSuite provides end-to-end metrics, uses large datasets, and accurately simulates popular online services, leveraging from Docker containers to ease the deployment of their content.

The general behavior of the workloads is analyzed by first running the benchmarks without specifying any CPU affinity, and it is observed that the benchmarks are executed on all active cores by default due to the OS being the entity in charge of distributing threads across nodes, without taking memory resources into consideration. Then, the architecture of the hardware is studied (an Intel® Xeon® Processor E5-4620 is used) to determine what is the best way to distribute memory resources across active threads of an application running by itself or with another Cloud workload.

Three possible memory resource scenarios are identified according to the selected architecture: all threads running on the same node (to study the behavior of the benchmarks

when both LLC and memory controller channel are shared) no sharing of LLC or memory channel (which means a maximum of 4 cores at a time should be used, since there are 4 available nodes), and LLC and memory controller sharing but by only two cores at a time.

Initially, each application is run alone on the server under the three mentioned memory resource scenarios. It is observed that how LLC and the memory controller channel are shared does in fact have an impact, but not on all applications, as previously noted in related work.

To further validate these findings and rule out a large enough cache that might misrepresent a contentious application as one that actually benefits from sharing, the applications are run alone again, this time with eight threads at a time, only to observe the same trend.

It is also verified that the impact in performance is due to memory resource sharing, by measuring L3 Hits and traffic towards memory.

Additionally, because there is more than one application at a time running in a server, two applications are placed running under the previous sharing scenarios, to find the optimal case for the application running alone will not necessarily be the same when there is a co-running benchmark.

Finally, based on the results from the experiments conducted in this work and a heuristic approach, a Scheduling Algorithm to assign threads to cores as optimally as possible, intended to run on top of the OS scheduler, is proposed.

CHAPTER TWO: BACKGROUND

Although a large group of servers performing parallel computations for diverse users [15] is also known as Datacenter, the underlying system powering the Cloud is not exactly that. The servers residing in Datacenters operate independently under the domain of different companies, have non-homogeneous hardware characteristics and run diverse types of small to medium applications.

Servers running Cloud Services on the other hand, are usually interconnected, grouped to offer a determined service (such as web searches or social networks), have similar hardware and software platforms, and belong under the same management domain. They are usually referred to as Warehouse Scale Computers (WSC).

Each server in the Cloud is powered by multiprocessor systems. As described in [13], computers are categorized, from a parallelism point of view, as either SISD (Single Instruction stream Single Data stream, also known as uniprocessor), SIMD (Single Instruction Multiple Data, meaning one instruction is executed by different processors using different data streams), MISD (Multiple Instruction streams, Single Data streams), and MIMD (Multiple Instruction streams, Multiple Data streams, each processor executes instructions and processes data independently). Most general purpose multiprocessors are based on MIMD because this model leverages thread-level parallelism.

2.1 Multiprocessors

A Multiprocessor is a type of computer composed by a tightly coupled group of processors that share a memory address space, running under the same operating system [13]. According to their memory organization, multiprocessors are divided in two groups:

1. Symmetric (shared-memory) multiprocessors (SMP, also referred to as Uniform Memory Access multiprocessors, UMA), which usually have a low number of cores, all of which will have symmetric access to a centralized memory.
2. Distributed Shared Memory multiprocessors (DSM), which are able to support a larger number of cores since the memory is distributed, allowing for more bandwidth and less latency. DSM are also known as Non-Uniform Memory Access multiprocessors (NUMA). Figure 1 illustrates a typical NUMA multiprocessor. It encompasses several multicore nodes interconnected via a network. Memory is shared across nodes, although the access time to the local portion of memory on each node will be less than the access time to a remote memory.

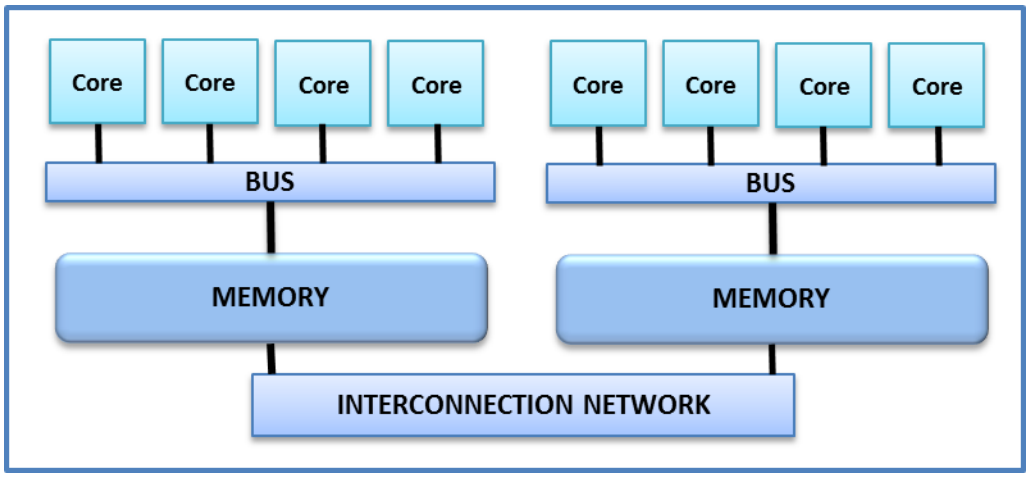

Figure 1: NUMA multiprocessor

One of the challenges in distributed memory systems is cache coherence. If a traditional snooping protocol were to be used, traffic towards memory would increase considerably, and the very purpose of distributing memory to allow for more bandwidth towards memory would be meaningless.

For this reason, NUMA multiprocessors sometimes use a directory protocol as an alternative to snooping-based cache coherence. However, snooping protocols are also used in multiprocessors; in section 2.2.2, an overview of MESI protocol is given, as this is the one implemented in the hardware used in this work.

2.2 MESI protocol

MESI (Modified, Exclusive, Shared, Invalid), is a multiprocessor invalidate protocol. It reduces bus traffic [17] by using write-back instead of write-through caches. The current state of each cache line (2 bits) can be as follows:

- *Modified:* the cache line has been modified and is not consistent with main memory (dirty).
- *Exclusive:* the cache line is consistent with main memory (clean) and is the only cached copy.
- *Shared:* the cache line matches main memory, but several copies of the line exist in other caches.
- *Invalid:* Line is not valid (no processor has it)

Figure 2 shows a State Transition Diagram for MESI. The dotted lines indicate transactions initiated by the bus. Continuous lines indicate transactions initiated by processor.

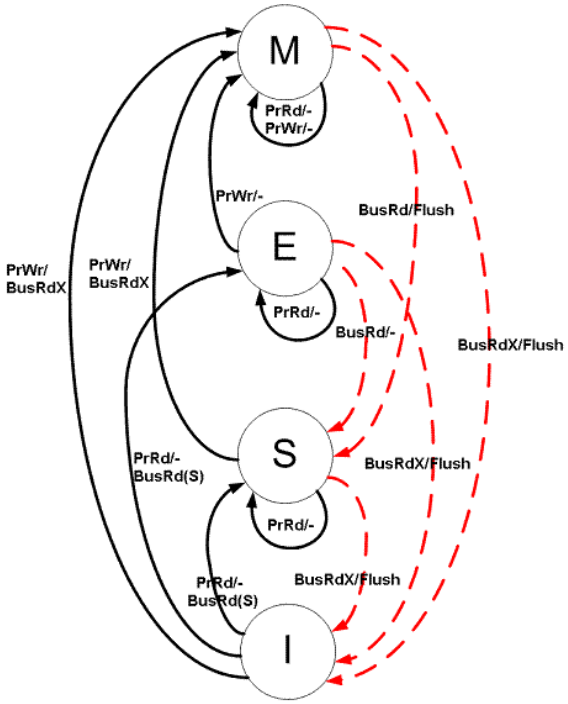

Figure 2: MESI protocol State Transition Diagram [21]

Each cache line will change its state as a function of memory accesses. The two possible operations are Reads and Writes to cache blocks. In the event of a Read miss, the block is loaded and its state will change to Exclusive if it was uncached in memory [19] or Shared if it was already shared, modified or exclusive.

If there is a Write miss, the block is loaded and marked Modified. All other cached copies are invalidated. On a Write Hit to an Exclusive or Shared block, the block is changed to Modified in both cases but invalidated for other cached copies only in the latter case.

2.3 Cloud Software

In general, Cloud characteristic applications have a high level of parallelism, handle large amounts of data, and are latency sensitive. They are often divided into offline and online services [14]. Online services comprise typical real-time transactions such as e-mail, web searches, etc., and offline services are used in large-scale data processing, either to generate a parallel

computation from a large dataset, such as graph processing or to analyze data to be used in an online application, such as processing satellite images for location services. Due to the scale-out nature of these applications, growth is addressed by adding more servers [16]. As a consequence, the performance of each individual server will shape the overall performance of the Cloud.

Unfortunately, the design of processors currently driving the Cloud was carried out with desktop scale-up workloads. As the Internet becomes more popular, a number of Cloud benchmarks have emerged, favoring scale-out behavior. In [15], an overview of the 4 most representative datacenter applications is presented: Web Search, Memory Caching, MapReduce, and Graph Analysis. A general overview of the survey is presented in this section.

- *Web Search:* The usual operation of a web search engine in a system with multiple servers starts by crawling, indexing and organizing data with a batch application. Web Search characteristic benchmarks adjust the size of the index so that it can be fitted into memory in order to avoid network and storage I/O. The index is divided evenly across servers, and then each server will use the index to calculate the relevance scores for each page before sending them to an aggregator which ultimately will produce the results for the user.
- *Memory Caching:* Due to the high latency of disk accesses, many popular services (such as Facebook) cache their non-media data in memory. One of the most popular frameworks for memory caching is Memcached, defined in [21] as *"an in-memory key-value store for small arbitrary data (strings, objects) from results of database calls, API calls, or page rendering"*. This arbitrary data is held in slabs inside the memory; both the data and slab size are determined by Memcached with the goal of increasing cache hit rates.

- MapReduce: the basic idea behind MapReduce is that computations are performed by splitting data across nodes. It consists of a run-time system, which is in charge of managing the deployment of the Map and Reduce functions specified by the programmer. The Map part applies a function to each section of the input data and produces a key-value pair, which is to be computed by the Reduce function to produce a result.
- Graph Analysis: Graphs are frequently used by online service providers to analyze user preferences and produce recommendations. In the case of social networks, Graphs are widely used to evaluate the influence of a user or an organization. Frameworks for graph processing allow asynchronous, dynamic, parallel computations, and task ordering according to data locality.

2.4 Related Work

Since Cloud computing has emerged as a utility in recent years, there is an increased effort in studying both the characteristics of Internet applications and hardware. Most studies have identified inefficiencies in modern hardware, due to the fact that server processors were designed with scale-up desktop applications in mind.

As explained in [1], the high instruction cache miss rate, low level of parallelism, and low requirements for bandwidth from scale-out applications, are some of the aspects that were not accounted for when designing server processors. They find that the instruction working sets of scale-out applications exceed both L1 and L2 capacity, but are smaller than LLC capacity. Moreover, LLC (which occupies more than half the die area) size beyond 6MB has minimal performance impact on scale-out workloads, due to their massive dataset size. They also point

out that because the complexity of Cloud applications is not taken into consideration, simple next-line prefetchers are not ideal for server multiprocessor systems.

In [3], server multiprocessor inefficiencies are also pointed out. The authors use some of the CloudSuite benchmarks to study the behavior of scale-out workloads on modern processors. They use Intel VTune to read performance counters and draw some micro-architectural conclusions, such as the high Instruction Cache miss rate from scale-out workloads, the low instruction level parallelism and memory-level parallelism from scale-out workloads, and also observe that bandwidth to memory is usually overprovisioned.

The PARSEC suite is used in [23], and it is shown that how threads are scheduled to share cache in different scenarios, has no impact on performance. In [4], the impact of memory resource sharing is studied for industry-strength datacenter applications; it is observed that thread to core mapping does have an effect, with up to 40% swings in performance for certain applications. Multiple thread-to-core mapping scenarios are considered in this publication, the ideal mapping for applications running alone on a server is identified, and it is observed that the optimal thread to core mapping will change depending on the co-running application. They propose three key characteristics of memory-application interaction that can determine the optimal thread-to-core mapping: Memory Bandwidth Usage, Data Sharing, and Cache Footprint.

CHAPTER THREE: CLOUDSUITE OVERVIEW

Since the Internet became an every-day life utility, the popularity of Cloud-based services such as web search, video streaming, social networks and multiple on-line shopping related transactions, has led to an increase in the demand for more robust servers to power Warehouse Scale Computers. Architectural decisions made for the design of processors being currently used in modern Cloud servers are based on traditional desktop computer benchmarking tools that mostly benefit scale-up workloads [1]

CloudSuite 3.0, a benchmark suite of emerging online services, provides real-world latency sensitive applications that handle large datasets to simulate common cloud massive data managing and gives an accurate performance assessment of modern server processors. It was designed based on a detailed michroarchitectural review of scale-out workloads.

By using performance counters, CloudSuite authors in [1] find that scale-out workloads have high instruction cache miss rates, have low instruction and memory-level parallelism, have low requirements of bandwidth and see no performance benefit from large LLC. They also analyze the most popular online services and find common characteristics across scale-out workloads:

- Scale-out run on large datasets that are divided into a large number of hosts.
- Requests served by scale-out workloads do not share any state.
- Account for unreliable machines in the underlying running infrastructure with specific application software.
- High-level task managing is handled with inter-machine connectivity.

Moreover, they find inefficiencies in the design of processors currently running Cloud applications. For instance, the high instruction cache miss rate from scale-out workloads cause

stalls, the oversized LLC adds significant latency (from fetching instructions from LLC), and the simple next-line prefetchers do not account for the complexity of Cloud applications, which as described by the authors in [1], are written in high-level languages, execute OS code, use third party libraries and have non-sequential access patterns.

CloudSuite aims at assessing the performance of modern hardware by representing real-world setups. Five CloudSuite benchmarks were selected for this work: Graph Analytics, In-Memory Analytics, Data Serving, Web Search, and Media Streaming. They are classified into Offline and Online Benchmarks. The following section comprises a high level overview of the functionality of these applications.

3.1 Offline Benchmarks

CloudSuite offline benchmarks [5] do not offer real-time metrics such as throughput or latency, but instead perform computations over large datasets. The performance criteria will be completion time.

3.1.1 Graph Analytics

The input of this benchmark is a large Twitter graph dataset that is processed using GraphX. The output of the benchmark is the influence of each user in the dataset, computed with PageRank. The job is distributed among all active or specified threads with significant communication across cores, due to the nature of the dataset. The performance metric is Running time, or the time it took the server to calculate the influence of each user.

3.1.2 In-Memory Analytics

This benchmark works as a recommendation system. The client container will take a dataset holding a large list of movies and will output a ranking based on an algorithm that uses matrix factorization to calculate user preferences. The performance metric is Running

(completion) time or the time it took for the application to compute recommendations. This benchmark will run in memory, and it is possible to allocate a certain amount of memory for each run, depending on the selected dataset size.

3.2 Online Benchmarks

CloudSuite online benchmarks provide real-time metrics, such as latency and throughput. These applications aim at simulating popular online services, such as streaming, search engines, and other types of online queries.

3.2.1 Data Serving

This application is based on YCSB (Yahoo! Cloud Serving Benchmark) which is basically a framework to measure performance of data store systems [6]. The objective of this benchmark is to simulate the behavior of popular online services that rely on NoSQL databases, such as airplane ticket reservations.

In a typical scenario where a reservation is to be made, the user will send a read request to a frontend server. A read request is then sent from the frontend server to a backend server, which will consult in a NoSQL database, and reply with the requested information. Once the user decides to proceed with the reservation, a write request will be sent to de frontend server and then from there to the backend server.

CloudSuite Data Serving models this behavior with a client container that will act as a request emulator and will send read and write requests to a backend server (a Docker container that starts a Cassandra server and mounts the dataset volume, held in a Docker container as well). A Docker network is also created to interconnect all the components. The output metrics for this benchmark are throughput (read and write operations per second) and latency.

3.2.2 Web Search

Based on the Apache Solr search engine framework, CloudSuite Web Search simulates a typical online search scenario, where a client sends requests to several index nodes with a determined query. In this case, a server Docker container will hold several index nodes images, and a client Docker container will send queries. A Docker network is used to interconnect all components. The output of this benchmark used in this work is throughput, given in ops/sec (search operations per second).

3.2.3 Media Streaming

This benchmark is comprised of a client based on httperf (traffic generator) that will send requests for video streaming to a server. A Docker container for each is created along with a dataset that contains several videos of different durations and qualities and a Docker network to interconnect all components. The performance metric is Net I/O (the streaming bandwidth) given in Kbps.

CHAPTER FOUR: SANDY BRIDGE OVERVIEW

The four node distributed shared memory processor used in this work (Intel Xeon E5-4620) is based on the microarchitecture code name Sandy Bridge. As explained in [7] it consists of a Front End unit in charge of fetching and decoding instructions into micro operations, a superscalar component capable of sending up to six micro-ops per cycle, and an in-order retirement unit that will ensure the results of the execution of the micro-ops follow program order.

Typically, the processor will search for the next instructions to execute, depending on the block selected by the Branch Prediction Unit, first on the Instruction cache and then progressing into the remaining levels of cache, L2 and L3, and then main memory. The generated flow of micro-ops is then dynamically scheduled and executed according to data order (instruction retirement will be consistent with program order).

4.1 Cache Hierarchy

There are three levels of cache in this architecture. In the first level, the L1 D (level 1 data cache) is "private" for each core, but can be shared in the event that multithreading is supported and enabled. There is also a L1 Instruction cache, with the same size as L1 DCache (32KB). L2 cache (256KB) is also dedicated to each core, but unified for instructions and data. LLC is interconnected to all cores in a node (also referred to as socket in this document) by a ring topology. For the Intel Xeon E5-4620, the LLC (L3) capacity is 16MB. L1 (both Data and Instruction), L2, and L3 are 8-way associative.

Snooping is managed across all levels of cache with MESI [8] (modified, exclusive, shared, invalid) cache coherence protocol. Each cache line on L1D, L2, and L3 has two MESI status

flags. Each of the flags will be marked according to the cache line state. Load and Store requests are managed in L1D.

4.1.1 Memory Reads

When data is required by an instruction, the core will read to L1, L2 and then L3 (memory if there is an L3 miss). All data contained in L2 and L1 should also appear in L3. Each L3 line is flagged to indicate which cores might hold that data, and if it is the case, L1D and L2 of a core that apparently has said data, will be looked up. If modified data is to be fetched, it will be referred to as a "dirty hit". If the data to be fetched is unmodified, it is called a "clean hit". In general, a clean hit will have lower latency than a dirty hit (60 cycles vs 43).

4.1.2 Memory Writes

In order to write to a memory location, the processor checks if it has the line on its L1D cache (in modified or exclusive MESI state). If the line is not found or is not in the right state, it will continue to search in the next levels of cache, caches from other cores, and memory. As soon as the line is in L1D, write will happen.

4.1.3 Last Level Cache

The LLC is divided and shared among available cores in the node. There is one portion of the cache assigned to each core. Each portion has one section for data coherency, memory ordering, LLC misses, writeback to memory, etc., and one section for cache lines (logic and data array sections respectively). LLC is interconnected to cores and memory controller via a ring topology (the bus encompasses four independent rings: data, request, acknowledge, and snoop, as described in [18]) and runs at the same frequency as the core clock, unlike previous generations.

4.1.4 Memory Controller

Accesses from the ring domain are managed by an arbiter, which is part of the Uncore (also referred to as System Agent). Memory traffic is forwarded from the arbiter to the memory controller, which supports multiple channels of DDR (4 in the Intel® Xeon® E5-4620, as illustrated in [18]), each channel being capable of running memory requests independently and contains an out of order scheduler that minimizes latency and maximizes memory bandwidth.

A write to the memory controller is considered completed when it is written to the write-data-buffer (each channel has a 32 cache-line write-data-buffer, which is flushed out to main memory at a different point in time, so that it does not affect write latency) [7].

CHAPTER FIVE: EXPERIMENTS AND RESULTS

The hardware used in this work is a four socket Intel® Xeon® E5-4620 processor. There are 8 cores per module and each core is capable of supporting up to two threads at a time. Only single threaded cores are used in this experiment. Each core has two levels: L1 and L2 of dedicated cache. L3 is shared between cores on the same node, with a "dedicated" portion of the cache for each core, which will be accessed faster by itself, but can also be accessed by other cores. The processor runs on a Dell PowerEdge R820 server with Ubuntu trusty 14.04 kernel release 3.19.0-25

The first part of the experiment consisted in testing each one of the five chosen CloudSuite Benchmarks (Graph Analytics, In-Memory Analytics, Data Serving, Web Search, and Media Streaming) running separately without specifying thread to core mappings. By default, the application threads are distributed by the OS scheduler among all active cores. To verify memory usage and core activity, the htop system monitor was run for all measurements.

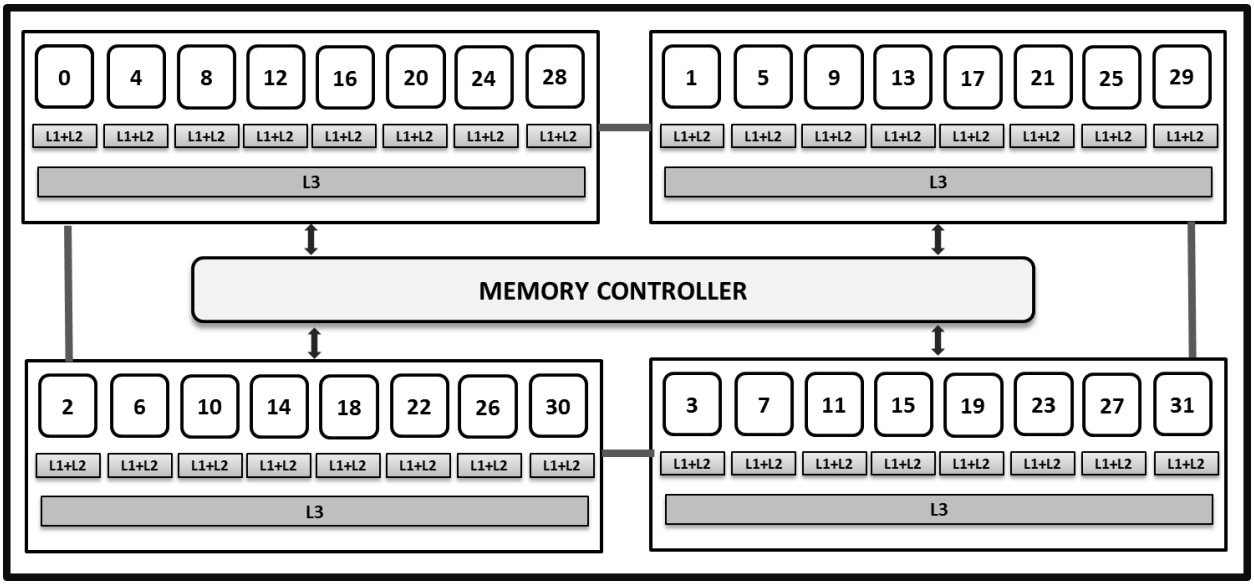

Figure 3: Intel® Xeon® Processor E5-4620 Memory Hierarchy overview

To assess the impact of memory resource sharing (either L3 or the memory controller channel on each node), and also consider the effect of running a cloud-based application simultaneously on different nodes without L3 sharing, four cores at a time were used. If 8 cores were to be used simultaneously, the scenario in which L3 and the memory controller channel are not shared across active threads of a given application could not be measured. Three different thread to core mapping scenarios were considered according to the architecture used in this work:

1. *One core per node*: 4 threads of the application running alone are distributed across the four available nodes, assigning one thread per core on each node. The objective of this setting is to assess the performance of each application when neither L3 nor memory controller channel are shared. It would be expected for contentious applications to be benefited from this configuration. Some degree of degradation should be perceived in non-contentious applications with significant data sharing.

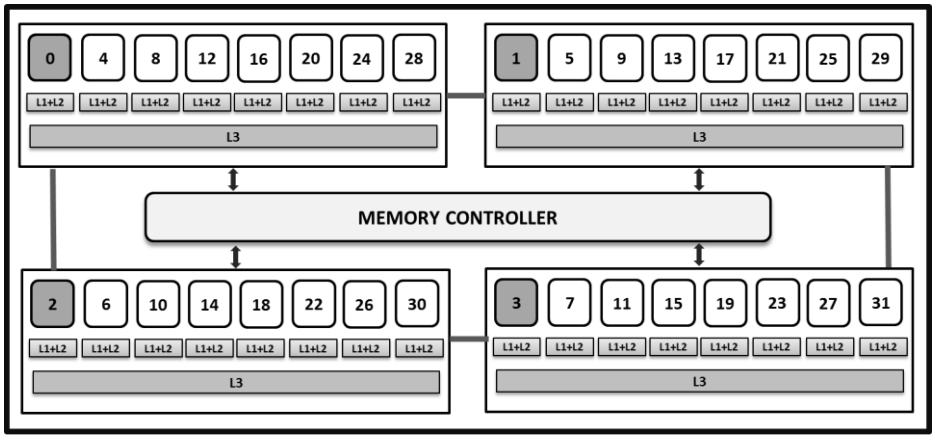

Figure 4: 4 Cores running simultaneously without sharing L3 or memory controller channel

2. *Two cores per node*: 4 threads of the application running alone are distributed across 2 nodes, with two cores per thread on each node. Both L3 and memory controller channel are shared in this setting; however, only by two threads at a time.

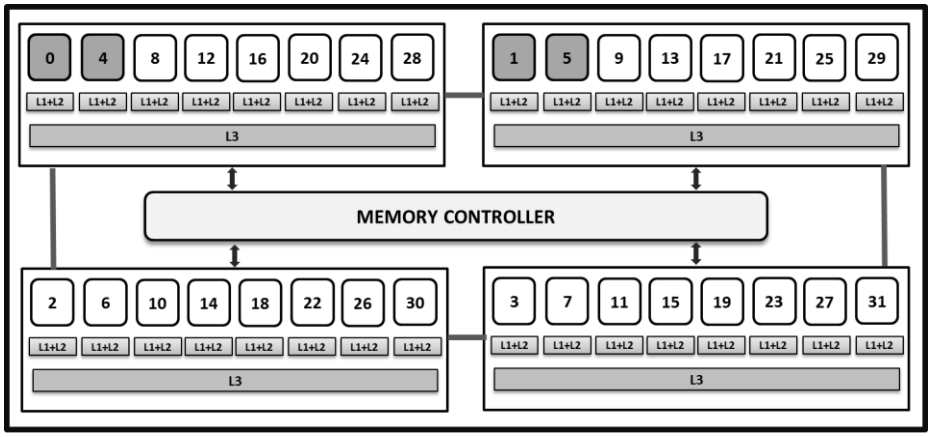

Figure 5: 4 Cores running simultaneously, L3 and memory controller channel are shared between two cores only.

3. Four cores per node: 4 threads of the active application running alone will be mapped to four cores on the same node. L3 and memory controller channel are shared by all used cores. The main difference between this configuration and the two cores per node setting is that more traffic through the memory controller channel is expected. Contentious applications should perceive degradation on this setting whereas applications with sharing of data should perform better.

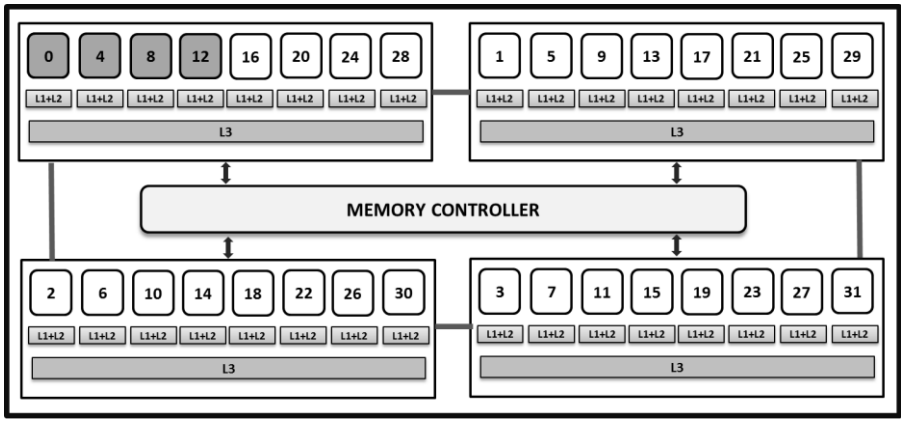

Figure 6: 4 Cores running simultaneously on the same node. L3 and memory controller channel are shared across all 4 cores.

The metric used to measure performance is the actual output of each benchmark. IPC is not considered since it is not a reliable indicator of performance in the case of multithreaded applications running on multiprocessor systems. As shown in [9], variations of IPC, either increase or decrease, not necessarily will reflect a performance improvement or detriment. The metrics used for each benchmark are enumerated in Table 1.

Table 1: Benchmark Metrics

Benchmark	Metric	Unit
Graph Analytics	Average Running Time	ms
In-Memory Analytics	Average Running Time	ms
Media Streaming	Average Net I/O	Kbps
Web Search	Throughput (search ops/s)	ops/s
Data Serving	Throughput - R/W	ops/s

5.1 Benchmark characterization

The process of running each benchmark requires downloading and running several Docker containers. In general, CloudSuite offline benchmarks require an image of the container that will run the benchmark and also a dataset. CloudSuite Online applications rely on a dataset, at least one server, one client and a network container to interconnect them. In this experiment, CPU affinity of the benchmarking Docker container is given passing the --cpuset-cpus <core IDs> argument. Core IDs are fetched from /proc/cpuinfo by using lscpu. The corresponding ID of cores is shown in Figure 3. Each benchmark was run in a loop of 10 iterations per configuration, using different core combinations according to each type of memory resource sharing scenario being tested. For instance, in the 4 Cores per Node case, the benchmark was run ten times on each node. Core combinations are shown in Table 2.

Table 2: Sets of cores for each 4 core test case

Configuration	Sets of cores
1 Core per Node	{0,1,2,3}, {4,5,6,7} {8,9,10,11}, {12,13,14,15}
2 Cores per Node	{0,4,1,5}, {0,4,2,6}, {0,4,3,7} {1,5,2,6}, {1,5,3,7}, {2,6,3,7}
4 Cores per Node	{0,4,8,12}, {1,5,9,13} {2,6,10,14}, {3,7,11,15}

For each of the core set, the benchmarks were run a total of ten times. The output of each run is then averaged per core configuration. Figure 7 illustrates the results for offline workloads. The performance metric is Average Running Time (ART, given in milliseconds) for both; a smaller average running time means better performance. Results for online benchmarks are shown in Figure 8. The metric for online benchmarks is throughput, meaning higher is better. In both cases, the results were normalized to the output measure of each benchmark running in the 1 Core per Node Configuration. To ensure threads were mapped as expected, the htop system monitor was used to verify CPU and memory usage.

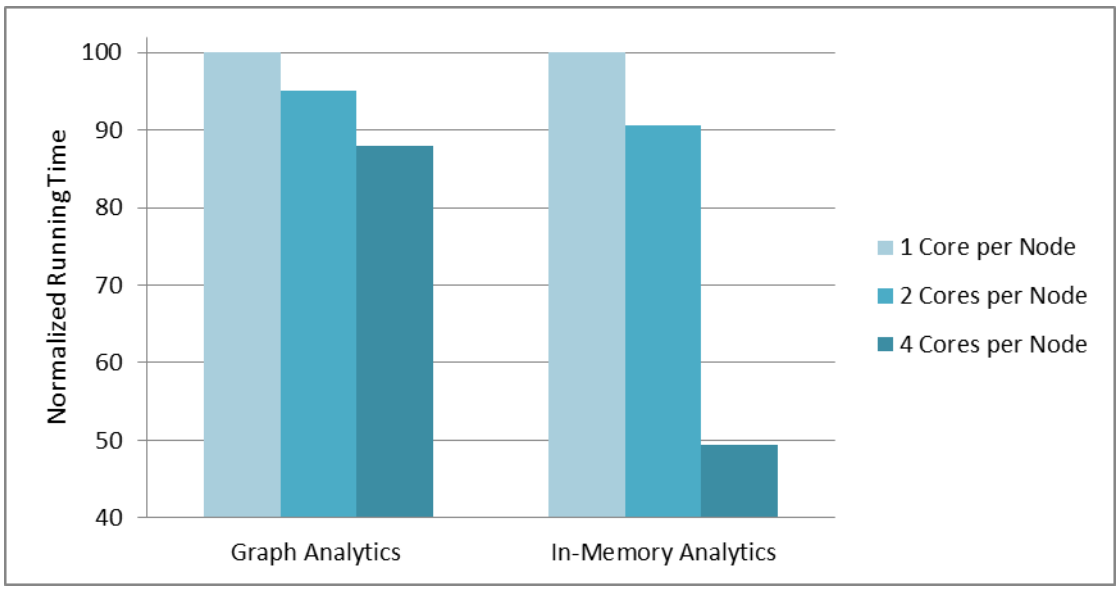

Figure 7: Performance results for offline CloudSuite Benchmarks: Graph Analytics and In-Memory Analytics

For offline benchmarks, there is a significant impact from running on different memory sharing scenarios. Both applications greatly benefit from sharing LLC and memory controller, meaning that there is significant sharing of data between active threads. The ART for Graph Analytics is 5% and 12.1% less on 2 and 4 Cores per node respectively than the ART for 1 Core per Node. The ART for In-Memory Analytics is 9.4% and 50.6% less on 2 and 4 Cores per node respectively than the ART for 1 Core per Node. Both applications show better performance when all of their threads are running on cores on the same node and degrade when no LLC or memory controller channel are shared.

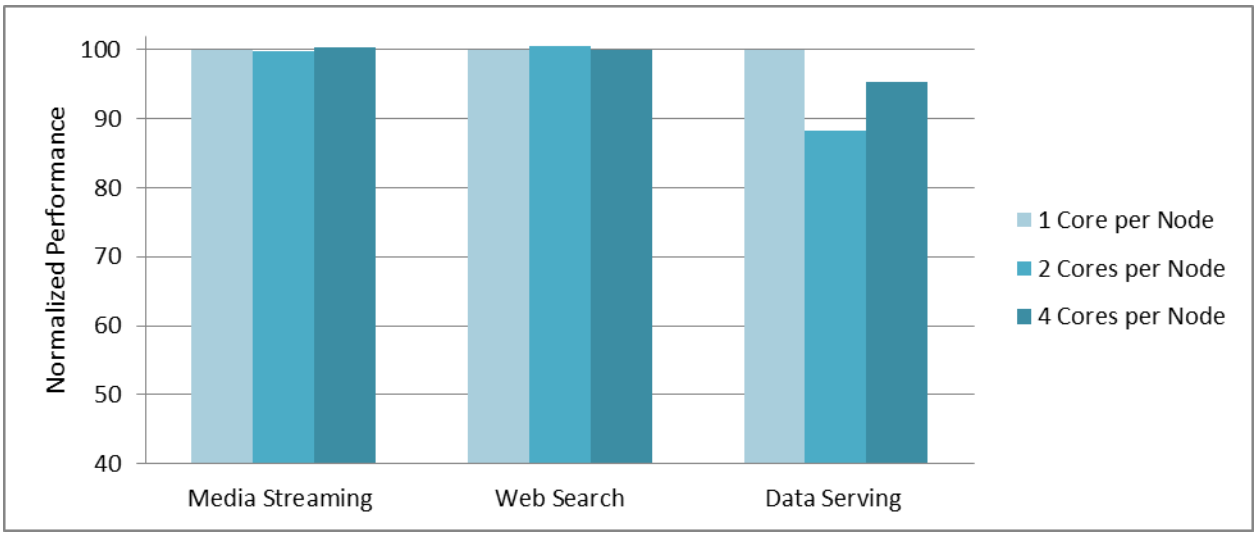

Figure 8. Performance results for online CloudSuite Benchmarks: Media Streaming, Web Search, and Data Serving.

The performance impact of changing memory resource sharing scenarios for CloudSuite online benchmarks on the other hand, with the exception of Data Serving, does not show any significant difference. There is a variation of less than 1% on the benchmark outputs (throughput).

As for Data Serving, a lot of contention for memory resources might explain the degradation of running this workload on the same node. The best performance is achieved when no LLC or memory controller channel is shared.

5.2 Eight Core Experiment

In the event in which cache is large enough to fit the data being processed by each benchmark, contentious applications might be mistakenly interpreted as non-contentious. To rule this possible scenario out, and also verify whether the impact of changing the configuration of memory resource sharing for Media Streaming and Web Search is actually negligible, the benchmarks were run using eight threads of each (mapped to eight cores). Core configurations are shown in table 3.

Table 3: Sets of cores for each 4 core test case

Configuration	Sets of cores
2 Cores per Node	{0,1,2,3,4,5,6,7}, {16,17,18,19,20,21,22,23} {8,9,10,11,12,13,14,15}, {24,25,26,27,28,29,30,31}
8 Cores per Node	{0,4,8,12,16,20,24,28}, {1,5,9,13,17,21,25,29} {2,6,10,14,18,22,26,30}, {3,7,11,15,19,23,27,31}

The same approach to testing as in the 4-core case is used, only this time sharing of resources is not completely ruled out. In the 2 Cores per node experiment, all four nodes will be used, meaning LLC and memory controller channel are going to be shared by only two cores, whereas in the 8 core case, all available cores per node will be active on one node at a time, and memory resources will be shared among them. Ten iterations of each combination are run, and the ART for offline applications and Throughput for online applications are averaged and normalized to the 2 Cores per Node case. Figure 9 shows the results for Graph Analytics and In-Memory Analytics and Figure 10 shows the results for Media Streaming, Web Search, and Data Serving.

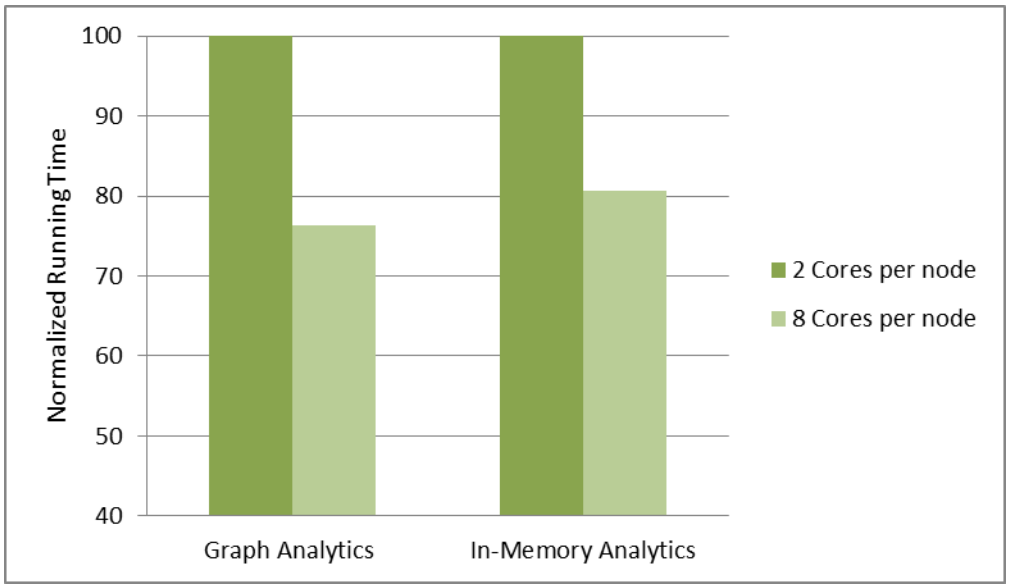

Figure 9: Performance results for offline applications running on 8 cores

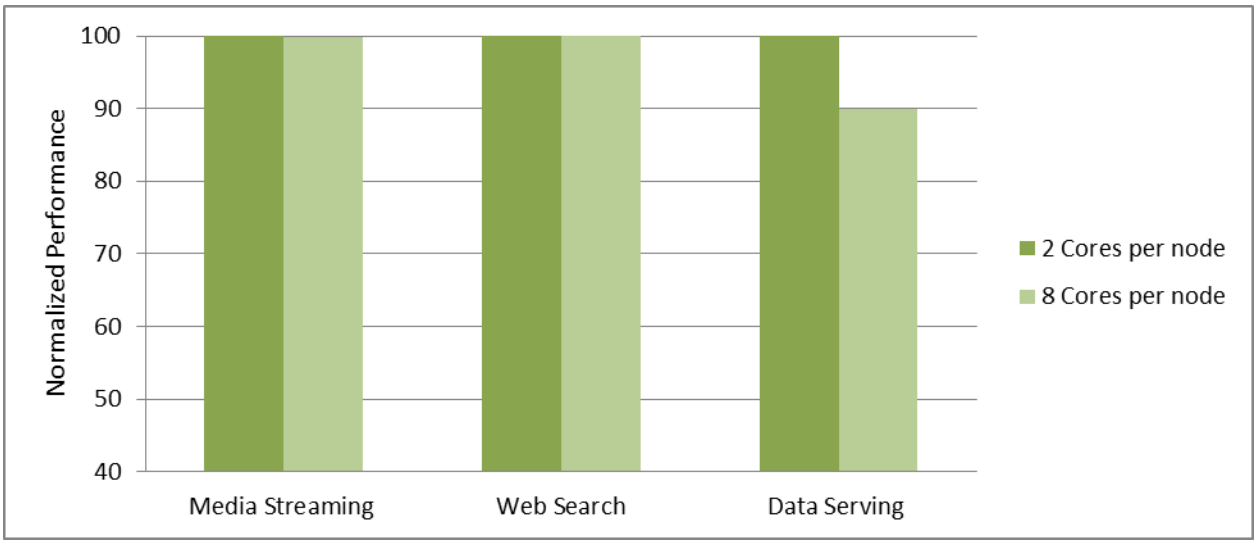

Figure 10: Performance results for online applications running on 8 cores

For the eight core scenario, the same thread as in the four core case is observed. Media Streaming and Web Search are not impacted at all. Graph Analytics and In-Memory Analytics benefit from sharing LLC and memory controller channel across all of their active threads. Data Serving on the other hand, proves to be a contentious application, benefiting from distributing its eight active threads across all four available nodes.

5.3 Performance counter monitor

In order to verify the effect of sharing memory resources one way or the other (1,2,4 Cores per node, as discussed previousely) in terms of LLC hit ratio and traffic towards memory, the Intel® Performance Counter Monitor was used. This application leverages from the performance counters on Intel processors (enabled by a set of Model-Specific Registers (MSRs) that can be programmed to report architectural performance monitoring events and are consistent across michroarchitectures [8]) to report core and uncore michroarchitectural information. The 2.11.1 version was run to measure L3 hits and memory traffic for each one of the selected benchmarks.

To make measurements as accurate as possible, all samples were taken when the applications were running alone on the server. One sample of each memory sharing scenario was used for each one of the workloads. Results from Intel® Performance Counter Monitor and the performance metric of the benchmark were documented according to the time recorded on the server timestamp. Once the Performance Monitor was started, the benchmark was then run for a given combination of cores (1,2,4 per node), the time of the beginning and ending of execution recorded, and then, results of the Performance Monitor (which were exported to a csv file) were compared to the previously saved times. The same procedure was then repeated for the next core combination. Only L3 Hit ratio and memory traffic are included in this report, as those are the resources that are being shared/not shared in the tests carried out in this work.

Since a comparison between the trend of the average performance of each benchmark from part 5.1 was necessary, the outputs of the benchmarks which are illustrated in Figures 11-15 were also analyzed.

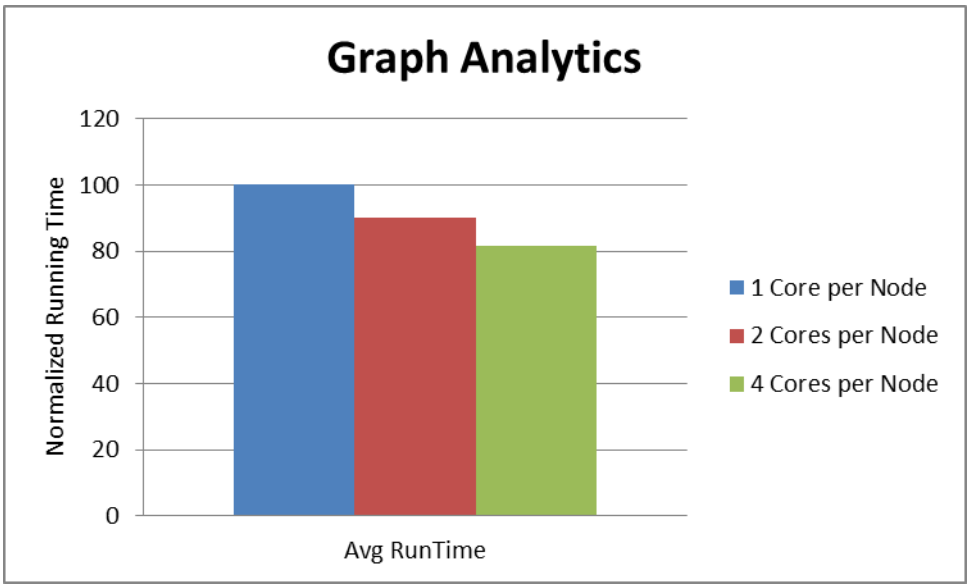

Figure 11: Graph Analytics normalized Running Time of sample used to take performance measurements

Table 4: Performance measurements for Graph Analytics

Graph Analytics Performance Monitor results			
Metric	**1 Core per Node**	**2 Cores per Node**	**4 Cores per Node**
Average Mem Traffic (GB/s)	0.61679084	1.077128742	1.578444
Average Mem Traffic per core (GB/s)	0.61679084	0.538564371	0.394611
Average L3 Hit ratio	0.273478836	0.33307755	0.369418667

For Graph Analytics, in Figure 11, Table 4, the trend initially observed is confirmed in this run. The application benefits from sharing threads on the same node, and degrades when its threads are split between nodes, with no sharing of memory controller channel or LLC. Performance monitor results show a correlation between the L3 hit ratio and memory traffic and the trend of the benchmark performance metric. The lowest Running time for the benchmark, was in the scenario of all threads running on the same node, which was also the case in which the L3 hit ratio was the highest (around 0.369, vs 0.333 and 0.273 in the 2 and 4 Cores scenarios respectively) and the traffic to memory was the lowest (0.394 GB/s, vs 0.539 GB/s and 0.617 GB/s in the 2 and 4 Cores scenarios respectively).

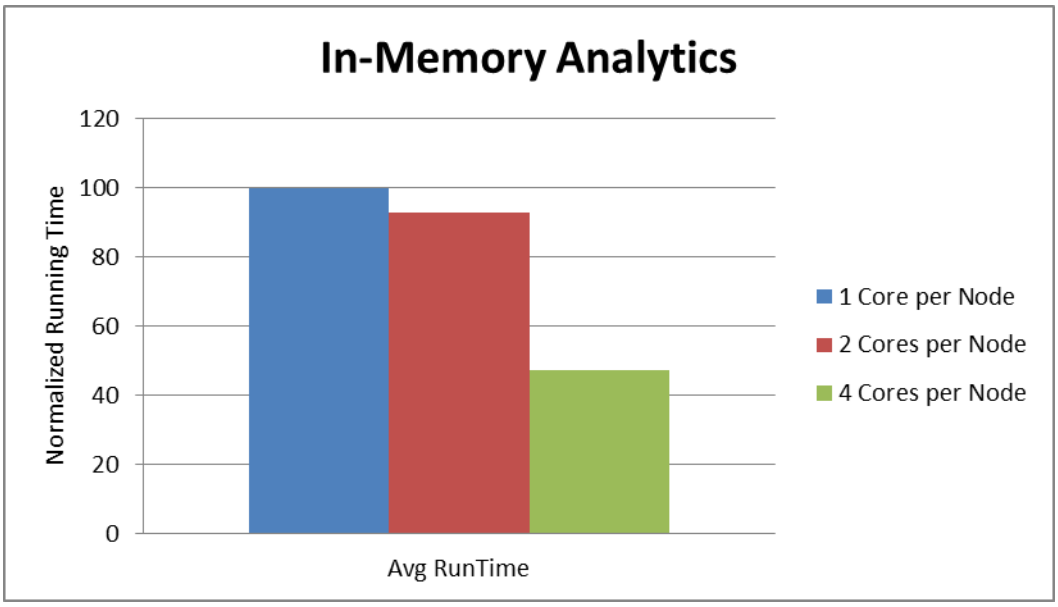

Figure 12: In-Memory Analytics normalized Running Time of sample used to take performance measurements

Table 5: Performance measurements for In-Memory Analytics

In-Memory Analytics Performance Monitor results			
Metric	**1 Core per Node**	**2 Cores per Node**	**4 Cores per Node**
Average Mem Traffic (GB/s)	0.265970703	0.480381704	1.5471875
Average Mem Traffic per core (GB/s)	0.265970703	0.240190852	0.386796875
Average L3 Hit ratio	0.520070182	0.578910326	0.630870573

In-Memory Analytics shows the same trend as was initially observed in 5.1 (Figure 12). A 52.8% improvement in Average Running Time is observed when comparing the 1 Core and 4 Cores per node case. This application also benefits from having all its threads on the same node, but unlike Graph Analytics, traffic to memory does not degrade performance. Table 5 shows the performance measurement results. Most of the computations for this benchmark must occur in L3 (as it can also be inferred from the application name), since in the 4 Cores per Node case the hit rate is around 0.631, and it decreases as sharing of L3 also decreases in the two following cases. However, traffic to memory is also the highest for the 4 Cores per Node case, but it does not affect the output of the benchmark.

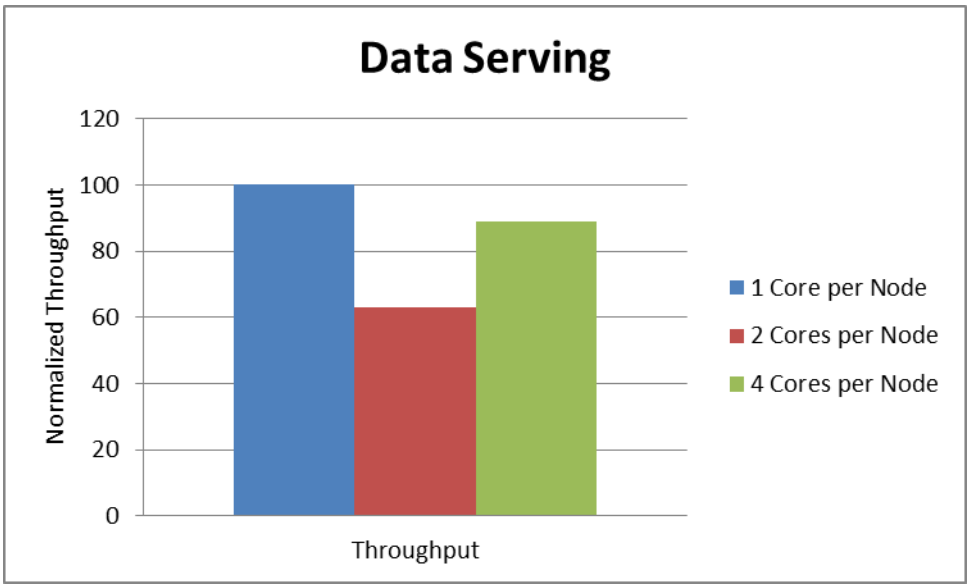

Figure 13: Data Serving normalized Throughput of sample used to take performance measurements

Table 6: Performance measurements for Data Serving

Data Serving Performance Monitor results			
Metric	**1 Core per Node**	**2 Cores per Node**	**4 Cores per Node**
Average Mem Traffic (GB/s)	0.047459167	0.0471845	0.11754
Average Mem Traffic per core (GB/s)	0.047459167	0.02359225	0.029385
Average L3 Hit ratio	0.6305	0.698475	0.7973

Contention of resources dominates the performance of Data Serving. As observerd in Figure 13 and Figure 8, the best throughput is obtained when running the application in the 1 Core per Node scenario. There is an interesting point about this trend: the best and worst performance are not given by either the 1 Core per Node or the 4 Cores per node like the previous two benchmarks. Although the best one is the 1 Core per Node scenario, the second best is the 4 Cores per Node scenario. This trend is set by the memory traffic, as noted in Table 6. The more traffic to memory, the better the performance will be. This actually is coherent with the metric of the application: read and write operations per second.

Another important point from Data Serving is that there appears to be some degree of sharing. The L3 hit ratio is the highest for the 4 Cores per node case, decreasing by around 10% when running on 2 Cores per Node and further degrading in the 1 Core per Node scenario. As noted in [3] both Data Serving and Web Search use a parallel garbage collector that artificially induces application level-communication, which explains why the L3 Hit ratio is better as more sharing of memory resources is present.

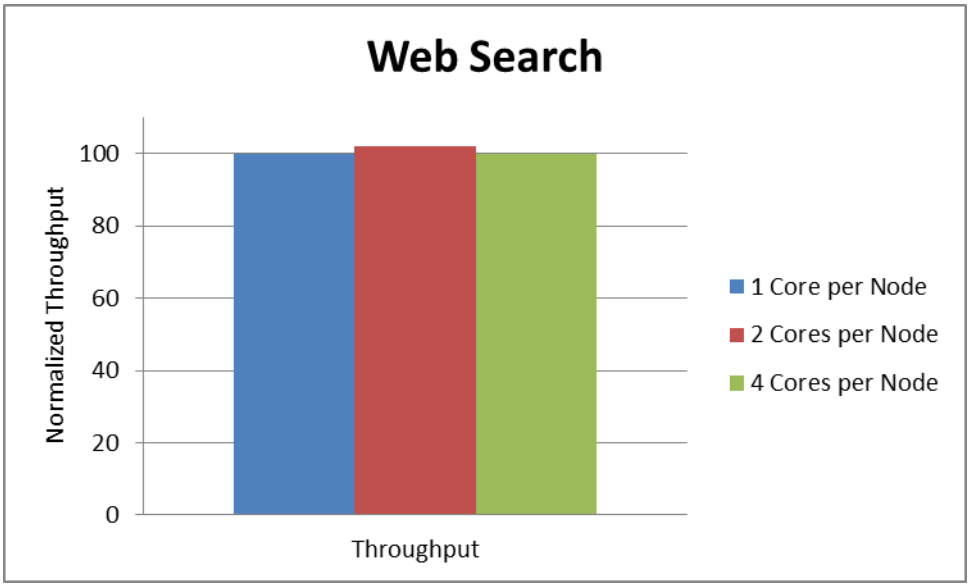

Figure 14: Web Search normalized Throughput of sample used to take performance measurements

Table 7: Performance measurements for Web Search

Web Search Performance Monitor results			
Metric	**1 Core per Node**	**2 Cores per Node**	**4 Cores per Node**
Average Mem Traffic (GB/s)	0.056893236	0.085877734	0.059960042
Average Mem Traffic per core (GB/s)	0.056893236	0.042938867	0.01499001
Average L3 Hit ratio	0.537934677	0.508305225	0.641316946

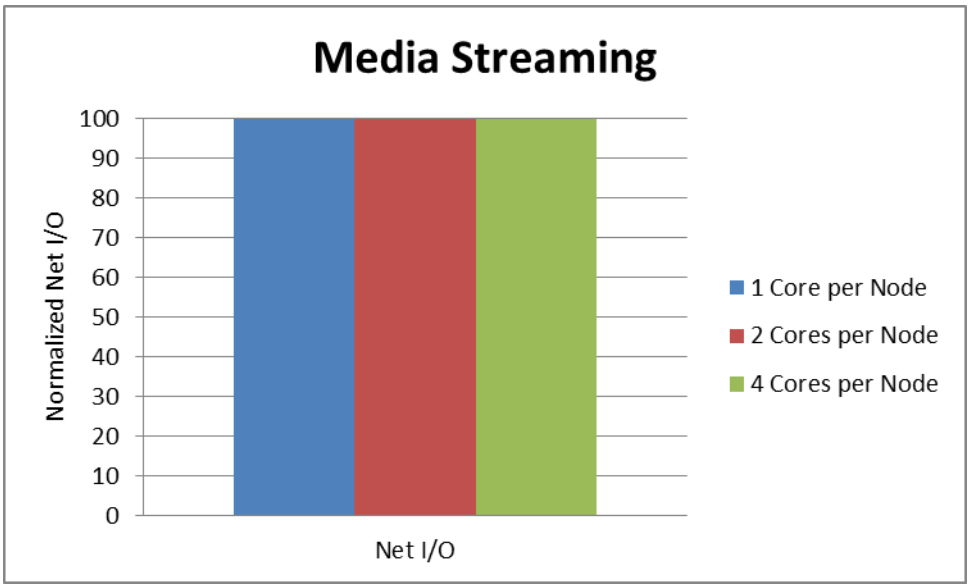

Figure 15: Media Streaming normalized Throughput of sample used to take performance measurements

Table 8: Performance measurements for Media Streaming

Media Streaming Performance Monitor results			
Metric	**1 Core per Node**	**2 Cores per Node**	**4 Cores per Node**
Average Mem Traffic (GB/s)	0.03744159	0.057086386	1.5471875
Average Mem Traffic per core (GB/s)	0.03744159	0.028543193	0.386796875
Average L3 Hit ratio	0.700281581	0.755926248	0.630164236

Web Search (Figure 13, Table 7) and Media Streaming (Figure 14, Table 8) are not impacted by how LLC and memory controller channel are shared. Even though it appears as if there is some sharing benefit, the garbage collector used in Web Search gives is responsible for such behavior. In the case of Media Streaming, it is also mentioned in [3] that the server used in this benchmark tracks the number of sent packages by updating global counters. For both Media Streaming and Web Search, less traffic to memory is observed when compared to In-Memory Analytics and Graph Analytics, which could indicate that L2 is sufficient for both applications.

An important observation from the performance monitor data in all Cloud-Based applications used in this work, is that memory bandwidth is underutilized. Traffic to memory did

not exceed 1.6 GB/s for any application (even when all active threads were running on the same node), even though the processor is provisioned for 42 GB/s according to technical specifications [22].

5.4 Performance of selected CloudSuite benchmarks with a co-running application

How applications will share resources is to be decided by the OS job scheduler. However, the OS scheduler does not take memory resources into account. Moreover, in order to guarantee QoS, latency sensitive applications are not usually co-located with non-latency sensitive applications [3], which results in an underutilization of processing resources.

The main goal in this part of the experiment is to study whether an offline application, co-running with an online application and viceversa will have an impact on the optimal thread to core mapping, assuming the best case scenario as the application running alone on the server with the same resource sharing configuration as in part 5.1.

5.4.1 Methodology

The same combinations of 1, 2, and 4 cores per node are used. A total of 4 cores on each case are considered. A total of 8 threads will be run at a time, 4 for each application. In the 1 core per node case, 4 threads of each application will be assigned to a core on different nodes. Application A (either online or offline, depending on each corresponding configuration) will have one assigned core per thread on each node, and application B (either online or offline, depending on each corresponding configuration) will be configured in the same way. In the 2 cores per node case, two threads of each application will be running on two cores per node, and they will be distributed on two nodes as well. Finally, for the 4 cores per node distribution, each application will have four threads mapped to four contiguous cores, all on the same node. Figure 16 shows configurations for each scenario.

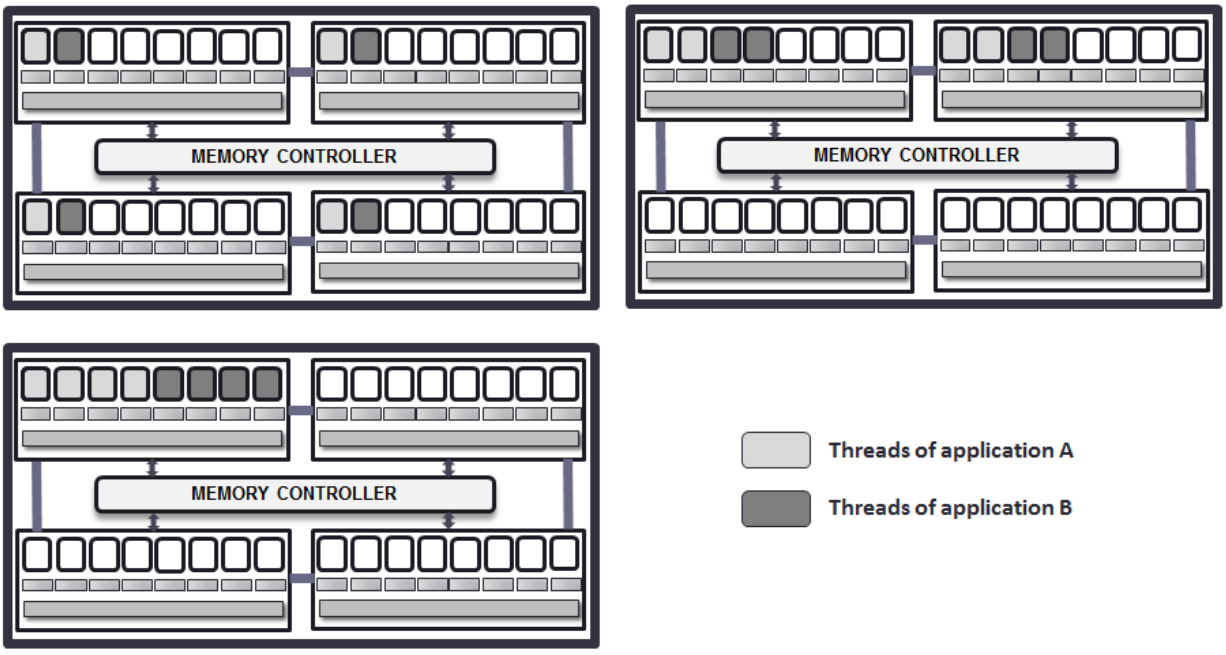

Figure 16: Thread to core mapping for co-running applications.

Each co-running scenario is normalized to the benchmark running on the 1 Core per Node configuration in the same scenario. Since the effect of assigning threads of Media Streaming and Web Search to different core configurations has a negligible effect on the performance of these applications (when running alone with either four or eitght threads at a time), it is also verified that the same result is achieved when placing them along with either latency sensitive or non-sensitive applications. In both cases, the variation in average trhoughput per configuration is less than 1.7%. Figures 17 and 18 illustrate this behavior.

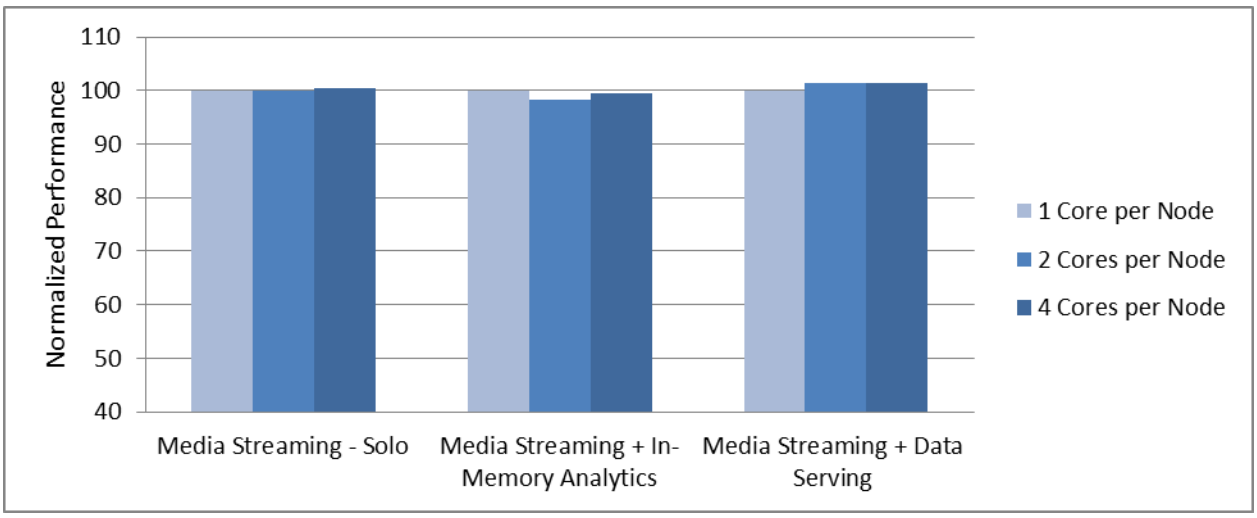

Figure 17: Media Streaming co-running with offline and online benchmarks. Normalized to 1 Core per Node configuration.

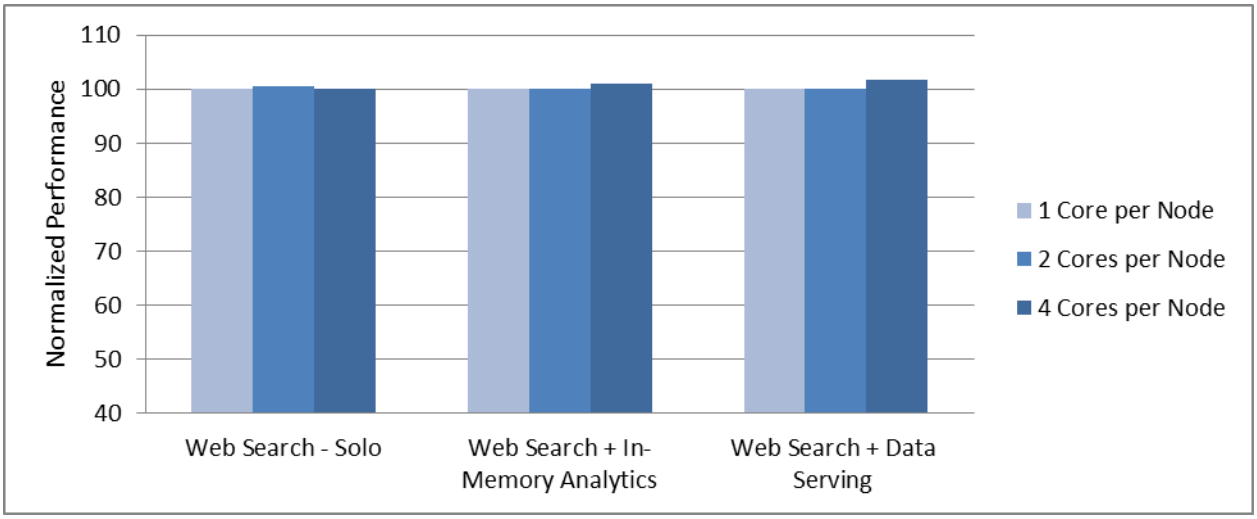

Figure 18: Web Search co-running with offline and online benchmarks. Normalized to 1 Core per Node configuration.

The offline benchmarks used in this experiment, Graph Analytics and In-Memory Analytics show the same behavior as in 5.1, as shown in Figure 19 and 20 respectively. Both benefit from having all of their threads in the same node, even if both are sharing the node with another application using the same number of threads. These applications degrade when their threads are running separately.

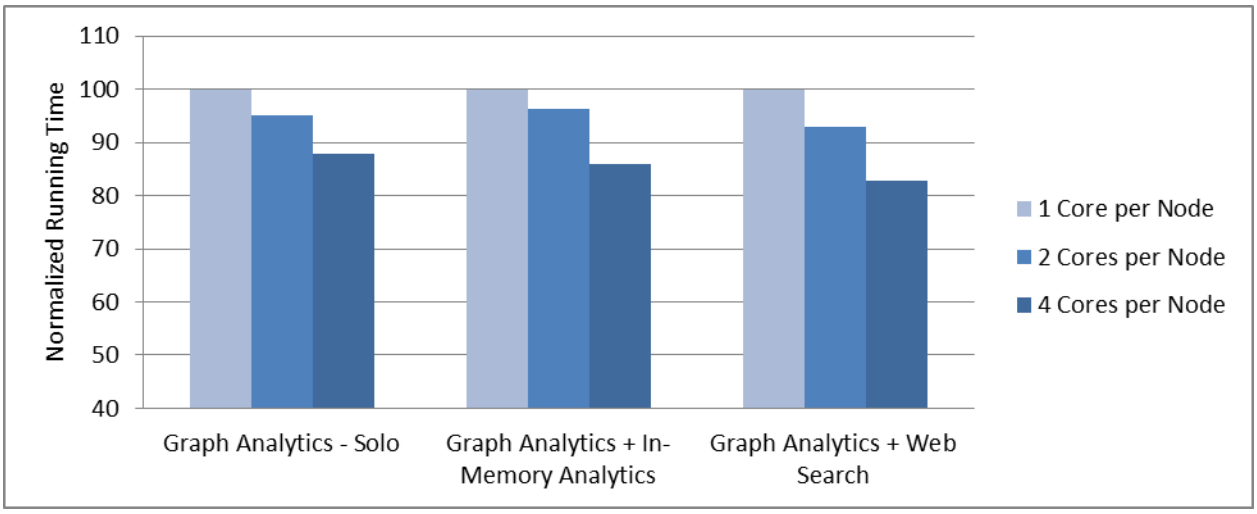

Figure 19: Graph Analytics co-running with offline and online benchmarks. Normalized to 1 Core per Node configuration.

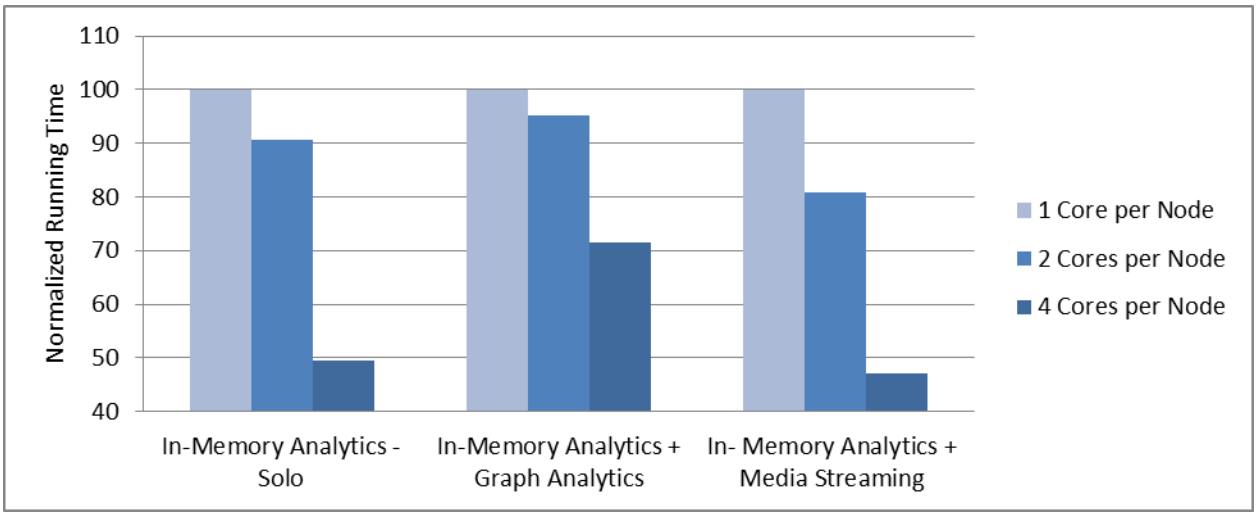

Figure 20: In-Memory Analytics co-running with offline and online benchmarks. Normalized to 1 Core per Node configuration.

The nature of the co-runner of Data Serving will change the ideal thread to core mapping of the application dramatically, as shown in Figure 21. When running solo, the best scenario was when running on separate nodes. However, if threads of In-Memory Analytics are added, the best mapping for this application is the 4 Cores per Node scenario. If the co-runner is an online application like Media Streaming, the optimal scenario will be the 2 Cores per Node one.

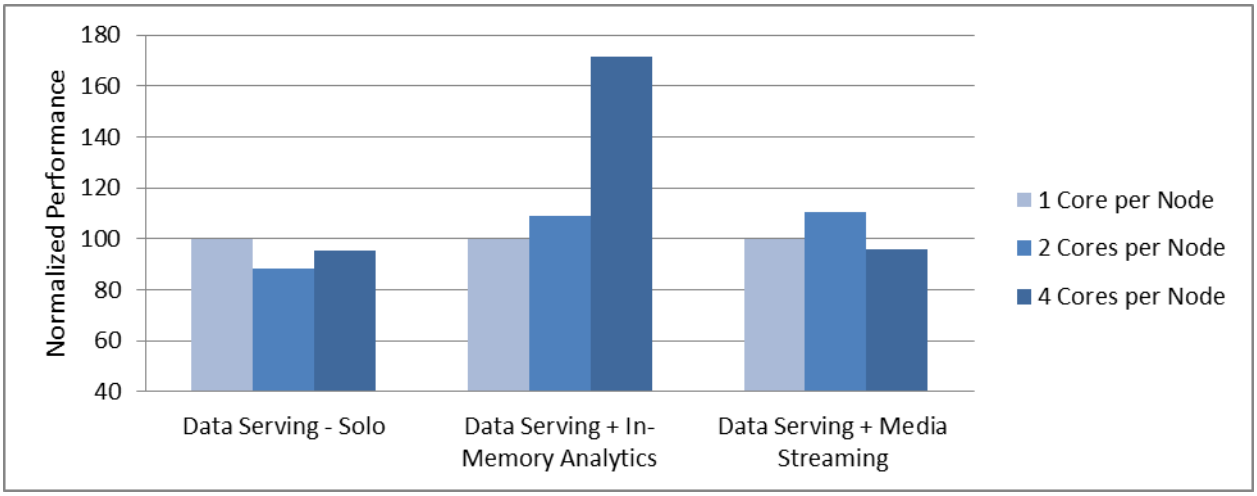

Figure 21: Data Serving co-running with offline and online benchmarks. Normalized to 1 Core per Node configuration.

CHAPTER SIX: A THREAD-TO-CORE SCHEDULING ALGORITHM

OS schedulers do not account for memory resource sharing or characteristics of the running application when assigning threads to particular cores, as noted in [4]. The OS scheduler focuses mainly on prioritizing cache affinity and load balancing. Even though threads can be assigned to a determined core or cores by the programmer, this will be dependent on the underlying architecture of the processor and the structure of the application.

In this section, an algorithm to schedule threads to cores according to the characteristics of each application, in terms of data sharing and how they benefit or degrade depending on how threads are distributed to either share or not LLC and memory controller channel is proposed. The impact of Co-Running applications is also taken into consideration, and the algorithm will produce an output for Cloud-Based applications running solo or with another Cloud-Based workload. This algorithm does not intend to replace the existing OS scheduler, but to provide an approach to optimize performance and memory resource usage on top of the Operating System, leveraging from CPU affinity to assign threads to cores.

The inputs to the algorithm are: the Application type, which can be either offline or online and the Co-Running Application type, which can also be offline, online, or none (meaning there will be no Co-Runner for the given application). Online applications can either be Read/Write (applications that issue both, read and write requests, such as Data Serving) or Read Only (applications similar to Web Search and Media Streaming, which only send Read requests and are not significantly affected by how memory resources are shared). For Read Only Online applications, the assigned mapping will be 2 Cores per Node, since there is some level of sharing (not as much as in the 4 Cores per Node case) while at the same time fewer nodes are used in this scenario compared to 4 Cores per Node.

The output of the algorithm will be the optimal memory resource sharing scenario (1,2 or 4 Cores per Node, same configuration as in section 5.4.1, Figure 16) for the requested application running alone or with another Cloud-Based application. The same configuration will apply to the Co-Running application, if there is one.

The possible values for the Application Type (defined as App_Type) are: Offline, Online_R, and Online_RW. For Co-Running Application Type, defined as Co_Run_Type, the values are: C_Online_RW, C_Online_R, C_Offline, and C_None. As for the output, defined as "Schedule" the possible values are 1_Core_per_Node, 2_Cores_per_Node, and 4_Cores_per_Node.

```
If App_Type = Offline then
   Schedule = 4_Cores_per_Node;
If App_Type = Online_R then
   Schedule = 4_Cores_per_Node;
If App_Type = Online_RW and Co_Run_Type = C_None then
   Schedule = 1_Core_per_Node;
If App_Type = Online_RW and Co_Run_Type = C_Offline then
   Schedule = 4_Cores_per_Node;
If App_Type = Online_RW and Co_Run_Type = C_Online_R then
   Schedule = 2_Cores_per_Node;
```

Figure 22: Proposed Algorithm

Figure 22 illustrates the proposed scheduling approach. Since Offline applications, Graph Analytics and In-Memory Analytics, or in general, applications that compute large amounts of data in a distributed fashion and produce a computation (which also have significant data sharing), the best scenario is to have all threads of the workload running in the same node, sharing LLC and memory controller channel, which is defined here as 4 Cores per Node.

As described in chapter 5, Online applications that have low LLC (L3 for the architecture used in this work) usage, little traffic to memory, and only send read requests to a remote server (Media Streaming and Web Search) will run mostly on L2. For that reason, the LLC and memory controller channel sharing will not impact performance in this case. Initially, the 2 Cores per Node was assigned to this particular case, mainly because if 1 Core per Node were to be used, the 4 available nodes would be occupied unnecessarily, and if the 4 Cores per Node were to be used, the memory controller channel in that particular node would have a slight but unnecessary increase in traffic. However, because these two online applications are not impacted by the sharing of LLC and memory controller channel, the performance for the co-running application was also taken into consideration in this case. Due to their high level of data sharing, Graph Analytics and In-Memory analytics degrade significantly when their threads are not running on the same node (More than 50% for In-Memory Analytics and around 18% when compared running on 1 Core per Node vs 4 Cores per Node), whereas the degradation for Data Serving when running on the 1 Core per Node configuration vs 4 Cores per Node configuration is around 5%.

In the case of Online applications that also send Read and Write requests to a remote server (such as Data Serving), the optimal memory resource sharing scenario varies depending on the Co-Runner. If the application is running alone, the 1 Core per Node scenario will be the best (because of the low level of data sharing and substantial contention of resources). If the Co-Runner is an offline application, the 4 Cores per Node scenario will be optimal. Finally, if the Co-Runner is an Online application that only sends read requests to a remote server, the optimal mapping will be 2 Cores per Node.

Table 9: Optimal Scheduling results from proposed Algorithm

Benchmark	Co-Running Application	Predicted Best Configuration	Optimal Configuration	Optimal Configuration (for co-Running Application)
Graph Analytics	Media Streaming	4 Cores per Node	4 Cores per Node	Any
	Web Search			Any
	In-Memory Analytics			4 Cores per Node
	Data Serving			4 Cores per Node
	No Co-Running Application			-
In-Memory Analytics	Media Streaming	4 Cores per Node	4 Cores per Node	Any
	Web Search			Any
	Graph Analytics			4 Cores per Node
	Data Serving			4 Cores per Node
	No Co-Running Application			-
Data Serving	Media Streaming	2 Cores per Node	2 Cores per Node	Any
	Web Search	2 Cores per Node	2 Cores per Node	Any
	Graph Analytics	4 Cores per Node	4 Cores per Node	4 Cores per Node
	In-Memory Analytics	4 Cores per Node	4 Cores per Node	4 Cores per Node
	No Co-Running Application	1 Core per Node	1 Core per Node	-
Web Search	Media Streaming	4 Cores per Node	Any	Any
	Graph Analytics			4 Cores per Node
	In-Memory Analytics			4 Cores per Node
	Data Serving			2 Cores per Node
	No Co-Running Application			-
Media Streaming	Graph Analytics	4 Cores per Node	Any	4 Cores per Node
	In-Memory Analytics			4 Cores per Node
	Web Search			Any
	Data Serving			2 Cores per Node
	No Co-Running Application			-

The prediction results from the algorithm are shown in Table 9. The predicted Best Configuration will apply to both, the main and co-running application. For the main configuration, all predicted scenarios are optimal. For the co-running applications, only the scenario of Data Serving co-running with either Web Search or Media Streaming would be sub optimal, with a ~5% decrease in throughput when compared to the 1 Core per Node configuration, as shown in Figure 21.

CHAPTER SEVEN: CONCLUSION AND FUTURE WORK

7.1 Conclusion

Five CloudSuite benchmarks were characterized according to how LLC and memory controller channel are shared.

In applications that are not latency sensitive, handle huge amounts of data and perform a computation as output, such as Graph Analytics and In-Memory Analytics, sharing dominates, meaning these applications will benefit if all of their active threads are sharing LLC and the memory controller channel.

In applications that perform read/write requests to a remote server such as Data Serving, the contention of resources dominates, meaning there is very little sharing of data, and for that reason, this applications will degrade if LLC and memory controller channel are shared.

For other latency sensitive applications, with relatively small datasets, that only send read requests to a remote server, L2 cache is sufficient, meaning how LLC and memory controller channel are shared, has a negligible effect on performance. It is also observed that there is less traffic to main memory from these two applications (Media Streaming and Web Search) when compared to other applications that run on large datasets.

Memory bandwidth in the used server Multiprocessor system is underutlized, as can be inferred from the Performance Monitor data in all Cloud-Based applications used in this work. Traffic to memory did not exceed 1.6 GB/s for any application (even when all active threads were running on the same node), even though the processor is provisioned for 42 GB/s according to technical specifications, meaning only up to 3.8% of the available bandwidth was occupied.

It is also noted that a co-running application can change the optimal memory resource sharing scenario for latency sensitive benchmarks that send read/write requests, but has little

impact in the optimal memory resource sharing of offline applications and latency sensitive applications that only send read requests to a remote server, regardless of their level of data sharing or their optimal mapping.

Finally, an algorithm (intended to run on top of the OS) that schedules threads to cores is proposed, based on each Cloud-Based application characteristics and the impact of running it under different memory resource sharing scenarios, either alone or with a co-running application.

7.2 Future Work

DDR3 is widely used in commercial servers and is optimized for bandwidth but not for energy efficiency. As noted in [11], 30% of the power consumption in Datacenters comes from DRAM; however, Cloud applications stress memory capacity and latency but use just a small percentage of the peak provisioned bandwidth [12].

Since memories designed for mobile platforms are optimized for low energy per bit, the impact of running Datacenter applications on LPDDR2 based servers vs DDR3 based servers is studied in [12]. Although significant energy efficiency improvements are observed, there is a small performance penalty. An interesting experiment would be to schedule threads of different Cloud-Based applications (according to their characteristics in terms of data sharing and how they benefit/degrade from memory resource sharing) in mobile-based servers, such that LLC and the memory controller channel are used effectively, achieving both, energy proportionality and improved performance.

APPENDIX A: INTEL® XEON® E5-4620 SPECIFICATIONS

Intel® Xeon® Processor E5-4620	
Lithography	32nm
# of Sockets	4
# of Cores per Socket	8
Max # of Threads per Socket	16
Processor Base Frequency	2.2 GHz
Max Turbo Frequency	2.6 GHz
VID Voltage Range	0.6V - 1.35V
Max Memory Size	384 GB
Memory Types	DDR3 800/1066/1333
Max Memory Bandwidth	42.63 GB/s
Instruction Set	64-bit
L1 Dcache	32 KB, 8-Way, 64-byte size line, Writeback
L1 Icache	32 KB, 8-Way
L2 Cache (Unified)	256 KB, 8-Way, 64-byte size line, Writeback
L3 Cache	16 MB, 16-Way, 64-byte size line, Writeback
L3 Cache bandwidth	192 bytes/clock

APPENDIX B: CLOUDSUITE CLIENT CONTAINER PARAMETERS

```
#Graph Analytics
      docker run  --cpuset-cpus=<cpu #s> --rm --volumes-from data
      cloudsuite/graph-analytics --driver-memory 16g
#In-Memory Analytics
      docker run  --cpuset-cpus=<cpu #s> --rm --volumes-from data_movies
      cloudsuite/in-memory-analytics /data/ml-latest-small
      /data/myratings.csv
#Data Serving
      docker run  --cpuset-cpus=<cpu #s> -it --name <clientname> --net
      serving_network cloudsuite/data-serving:client cassandra-server
#Web Search
      docker run  --cpuset-cpus=<cpu #s> -it --name <clientname> --net
      search_network cloudsuite/web-search:client 172.22.0.2 50 90 60 60
#Media Streaming
      docker run --cpuset-cpus==<cpu #s> -t --name=<clientname> -v
      $path1:/output --volumes-from streaming_dataset --net streaming_network
      cloudsuite/media-streaming:client streaming_server
```

REFERENCES

[1] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., . . . Falsafi, B. (2012). Clearing the clouds: A study of emerging scale-out workloads on modern hardware. *ACM SIGARCH Computer Architecture News, 40*(1), 37-48. doi:10.1145/2189750.2150982

[2] Palit, T., Shen, Y., & Ferdman, M. (2016). Demystifying cloud benchmarking. Paper presented at the 122-132. doi:10.1109/ISPASS.2016.7482080

[3] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., . . . Falsafi, B. (2014). A case for specialized processors for scale-out workloads. *IEEE Micro, 34*(3), 31-42. doi:10.1109/MM.2014.41

[4] Tang, L., Mars, J., Vachharajani, N., Hundt, R., & Soffa, M. L. (2011). The impact of memory subsystem resource sharing on datacenter applications. *ACM SIGARCH Computer Architecture News, 39*(3), 283-294. doi:10.1145/2024723.2000099

[5] "Data Serving · CloudSuite." Available: http://cloudsuite.ch///pages/benchmarks/dataserving/.

[6] "2016-CloudSuite-EuroSys-Tutorial.pdf." Available: http://cloudsuite.ch//public/presentations/2016-CloudSuite-EuroSys-Tutorial.pdf.

[7] "Intel® 64 and IA-32 Architectures Optimization Reference Manual" Available: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html

[8] "Intel® 64 and IA-32 Architectures Software Developer's Manual" Available: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[9] Alameldeen, A. R., & Wood, D. A. (2006). IPC considered harmful for multiprocessor workloads. *IEEE Micro, 26*(4), 8-17. doi:10.1109/MM.2006.73

[10] Mars, J., Vachharajani, N., Hundt, R., & Soffa, M. L. (2010). Contention aware execution: Online contention detection and response. Paper presented at the 257-265. doi:10.1145/1772954.1772991.

[11] "Energy-Efficient Data Centers and Systems." Available: https://pdfs.semanticscholar.org/3e24/0e06d79bf43ece6504be3d3b22e5e08afb06.pdf.

[12] Malladi, K., Lee, B., Nothaft, F., Kozyrakis, C., Periyathambi, K., & Horowitz, M. (2012). Towards energy-proportional datacenter memory with mobile DRAM. *ACM SIGARCH Computer Architecture News, 40*(3), 37-48. doi:10.1145/2366231.2337164

[13] Hennessy, J. L., & Patterson, D. A. (2014). *Computer architecture: A quantitative approach* (4th ed.). Burlington: Elsevier Science.

[14] Barroso, L. A., Clidaras, J., & Hölzle, U. (2013). *The datacenter as a computer, 2nd edition* (2nd ed.) Morgan & Claypool Publishers.

[15] Lee, B. C. (2016). *Datacenter design and management: A computer architect's perspective*. San Rafael, California (1537 Fourth Street, San Rafael, CA 94901 USA): Morgan & Claypool.

[16] Lotfi-Kamran, P., Grot, B., Ferdman, M., Volos, S., Kocberber, O., Picorel, J., . . . Falsafi, B. (2012). Scale-out processors. Paper presented at the 500-511. doi:10.1109/ISCA.2012.6237043

[17] "Mesi.pdf." Available: http://www.cs.utexas.edu/~pingali/CS377P/2017sp/lectures/mesi.pdf.

[18] "Intel® Xeon® Processor E5-2600/4600 Product Family Technical Overview | Intel® Software." Available: https://software.intel.com/en-us/articles/intel-xeon-processor-e5-26004600-product-family-technical-overview#_Toc341794173.

[19] "Coherence.pdf." Available: https://people.cs.pitt.edu/~melhem/courses/xx45p/coherence.pdf.

[20] "MESI Protocol." Wikipedia, March 3, 2017. https://en.wikipedia.org/w/index.php?title=MESI_protocol&oldid=768301337.

[21] "Memcached/Memcached." GitHub. Available: https://github.com/memcached/memcached.

[22] "Intel® Xeon® Processor E5-4620 (16M Cache, 2.20 GHz, 7.20 GT/S Intel® QPI) Product Specifications." Intel® ARK (Product Specs). Available: http://ark.intel.com/products/64607/Intel-Xeon-Processor-E5-4620-16M-Cache-2_20-GHz-7_20-GTs-Intel-QPI.

[23] Zhang, E., Jiang, Y., & Shen, X. (2010). Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? Paper presented at the 203-212. doi:10.1145/1693453.1693482

VITA

Ana Bernal Montana is a graduate student at the University of Texas at San Antonio. She is pursuing a Master of Science degree in Electrical Engineering and currently works as a System Validation Engineer at Intel Corporation. She is originally from Colombia, where she earned a Bachelor's degree in Electronics Engineering at Universidad Santo Tomas, Bogota. Her future plans include trying for a PhD in Computer Engineering, exploring research opportunities, and eventually becoming a Computer Architect.