Memory System Evaluation for Disaggregated Cloud Data Centers

by

Andreas E. Andronikakis

A DIPLOMA THESIS

submitted to

Technical University of Crete

Presented May 8, 2017

This thesis is dedicated to my parents.
For their endless love, support and encouragement.

ABSTRACT

In our time, there is a plethora of highly demanding computational work and of major scientific research and applications. Various ways of compressing production costs for the above operations are also sought. There can be found a variety of efforts to meet the above-mentioned needs such as those that promote the automation of production, those which, more generally, seek to reduce operating costs, but simpler and less effective are those that seek to reduce non-functional costs. Such a practice and prospect could include the development of cloud computing and, more specifically of cloud servers, which is a necessary infrastructure for cloud computing services to work.

Their ease of use, scalability, low cost, and reliability are some of the most important reasons for deploying cloud computing, cloud servers and their respective service providers. One element to be highlighted is the need for a high-standard memory in terms of performance and size. The existing architecture of Cloud Data Centers is characterized by high energy consumption and a great waste of resources (mainly memory).

This thesis refers to estimating the memory of cloud data centers with disaggregated (or disintegrated) servers, ie servers whose components and/or resources are in separate sub-assemblies, regarding their physical location. This type of servers is being researched over the past years, and its advantages and disadvantages are examined. This Disaggregated Architecture System aims to change the traditional way of organizing a Data Center by proposing a more flexible and software-modulated integration around blocks, the Pooled Disaggregated Resources, as opposed to traditional unification around from the mainboard.

The purpose of this diploma thesis is to study and develop a unified (modular) memory simulation tool of the above architecture, driven by the execution of an application, the **DiMEM Simulator** (DIsaggregated MEMory System Simulator). This simulator was implemented to approximate the behavior of standard Cloud application workloads in Shared Architecture Memory. The tool combines the Intel Pin Framework, where this diploma focuses, with the DRAMSim2 memory simulator.

The subject of this thesis is the study of the Dynamic Binary Instrumentation, the understanding of Cache levels and their simulation methods, the implementation of the Disaggregated Architecture Memory simulation, and experimentation with various parameters. The results presented approximate the overall behavior of a Memory System of a Disaggregated Cloud Data Center.

TABLE OF CONTENTS

Page

1 Introduction 1

1.1 Thesis Contributions 4

1.2 Thesis Outline 5

2 Theoretical Background 7

2.1 Instrumentation 8

2.1.1 Source Instrumentation 9

2.1.2 Binary Instrumentation 9

2.1.3 Dynamic Instrumentation 10

2.1.4 Static Instrumentation 12

2.2 Intel PIN : A Dynamic Binary Instrumentation Tool 13

2.3 Disaggregated Architecture 18

2.3.1 Disaggregated System Design 19

2.3.2 Replace electrons with photons 20

3 Related Work 23

4 Implementation of DiMEM Simulator 26

4.1 Front-End of DiMEM Simulator: Instrumentation stage 26

4.1.1 Functional simulation of TLB and cache hierarchy using PIN 26

4.1.2 Instrumentation Function 31

4.1.3 Analysis Functions 33

4.1.4 Instrumentation Output 36

4.1.5 Multithreading Functionality 38

4.1.6 Hyperthreading Functionality 41

4.2 Back-End of DiMEM Simulator: Simulation stage 42

5 Evaluation and Experimental Results 43

5.1 Evaluation 43

5.1.1 CPU Metrics 43

5.1.2 DRAM Metrics 45

5.1.3 Cloud Benchmarks 46

5.2 Experimental Process 48

5.2.1 Data Caching Profiling and Utilization 50

5.2.2 Simulation Scenarios 51

5.3 Experimental Results 52

TABLE OF CONTENTS (Continued)

Page

5.3.1 Dense Data Caching Experimental Results 53

6 Conclusion and Future Work 69

6.1 Conclusion 69

6.2 Future Work 71

Appendices 73

A Some useful codes 74

B Simulation output 86

LIST OF FIGURES

Figure Page

1.1 Whole System Abstract Figure. 4

2.1 This diagram depicts a scheme pertaining to the dynamic type (DBI). 10

2.2 DBI Frameworks. 11

2.3 This diagram depicts a scheme pertaining to the dynamic type (DBI). 13

2.4 The architecture of Pin. 14

2.5 1 Trace, 2 BBLs, 6 instructions 17

2.6 Architectural differences between server-centric and resource-centric datacenters. 18

2.7 Block Diagram of high-level dReDBox Rack-scale architecture . . . 20

2.8 Disaggregated data center interconnect requirements (Courtesy of The Optical Society) [7] 21

4.1 The DiMEM Simulator Logo. 26

4.2 Cache hierarchy of the K8 core in the AMD Athlon 64 CPU. 27

4.3 An example of a Multicore cache hierarchy 29

4.4 local.hpp File Reference 30

4.5 Front-End Diagram 31

4.6 Analysis Function 33

5.1 DataCaching Power8 workload analysis. 51

5.2 Power 8 DataCaching 2GB Results 54

5.3 Cortex-A53 DataCaching 2GB Results 55

5.4 Cortex-A57 DataCaching 2GB Results 56

5.5 Power 8 DataCaching 4GB Results 57

5.6 Cortex-A53 DataCaching 4GB Results 58

5.7 Cortex-A57 DataCaching 4GB Results 59

LIST OF FIGURES (Continued)

Figure		Page
5.8	Power 8 DataCaching 8GB Results	60
5.9	Cortex-A53 DataCaching 8GB Results	61
5.10	Cortex-A57 DataCaching 8GB Results	62
5.11	Power 8 DataCaching 16GB Results	63
5.12	Cortex-A53 DataCaching 16GB Results	64
5.13	Cortex-A57 DataCaching 16GB Results	65
5.14	Power 8 DataCaching 32GB Results	66
5.15	Cortex-A53 DataCaching 32GB Results	67
5.16	Cortex-A57 DataCaching 32GB Results	68

LIST OF TABLES

Table — Page

4.1 Multithreading example 41

5.1 CPU table 1 44

5.2 ARM Cortex-A53 Cache Metrics 44

5.3 ARM Cortex-A57 Cache Metrics 44

5.4 IBM Power8 Cache Metrics 44

5.5 DRAM Metrics - Modified DDR4 Micron 1G 16B x4 sg083E 45

5.6 Nanosecond to Memory Cycles Disaggregated Latency Correspondence 52

CHAPTER 1

Introduction

"We believe we're moving out of the Ice Age, the Iron Age, the Industrial Age, the Information Age, to the participation age. (...) You are participating on the Internet, not just viewing stuff. We build the infrastructure that goes in the data center that facilitates the participation age."

— Scott McNealy, former CEO, Sun Microsystems

Cloud computing is where software applications, data storage and processing capacity are accessed from a cloud of online resources. This permits individual users to access their data and applications from any device, as well as, allowing organizations to reduce their capital costs by producing software and hardware as a utility service[9]. Cloud computing is closely associated with Web 2.0.

These times, as was in the past, organizations, institutes, even legal entities engage in the business and production sector, looking for various ways to boost output, improve productivity and compress production costs. One can find diverse efforts, such as those that promote the automation of production; in general, they seek to reduce operating costs, but being simpler and with less impact than those that seek to reduce non-recurring costs. The development of cloud computing could be included in such a producting practice and perspective, as well as, more specifically, cloud servers, which consist a necessary infrastructure enabling one to operate the cloud computer services. This is a booming field that has attracted providers like Microsoft and Amazon, which have developed Microsoft Azure and Amazon Web Services (AWS), respectively. The main services offered are the IaaS (Infrastructure as a Service), PaaS (Platform as a Service), SaaS (Software as a Service), i.e. infrastructure services, platform and software[4].

One issue that arises is the specific booming and growth causes in cloud computing over the past years, with greater emphasis on those from 2009 onwards. The thesis will list and analyze some of the most fundamental ones below[4]:

1. **Ease of use**; in contrast to the "more traditional" software development environments, the cloud servers provide a range of user interface tools which make development codes much easier and "comfortable". Such user interface tools are the CLI and web CLI, the web UI, various API and IDE.

2. **Scalability**; scalability is defined as "the ability of the system for easy extension by adding more machines, and vice versa, without influencing the existing setup." This advantage is related to the action and to the capacity that the cloud server services have, in terms of cost. (i.e. all services normally charge pay-as-you-go, which correspond exactly to the type and temporal extent of provision of selected services).

3. **Business or organizational risk**; the cloud computing is considered to ,greatly , reduce the risk that a company assumes based or heavily using Information Technology (IT), as long as the appropriate infrastructure nary

fixed costs of the business, due to the coverage of the need for computing infrastructure through the cloud.

4. **Cost**; as mentioned above, similar service providers through cloud servers correspond exactly to the demand required by each specific customer, while the burden is only on the basis of accurate computational power and time use of specific computing resources. Given that, the potential for cloud computing services to the type, quantity and time, required one need to be of no capital expense texture but only of functional. At the same time, there is no need for maintenance of servers or software, and therefore thus, the total cost of the organization or undertaking is reduced.

5. **Reliability**; is a key element that justifies the development of cloud computing and, for some, it is a primary key element. What is indicated in these cases is that storing data in a portable storage medium such as a USB Stick disk or CD / DVD is a less reliable solution than storing them in a cloud computing service, such as, for example, Rapidshare, Mega or the Dropbox.

All the above explains the development of cloud computing, the cloud server and the respective service providers. One point that should be emphasized in this introduction is the need for a high standard memory. An explanation given to the urgency of the existence of this element from the inside to the lowest extent is that Cloud Computing services (IaaS) meet virtual memory services, as well. But an even more important issue has to do with the fact that cloud computing services are memory intensive. In other words, such an architecture (and amount) of memory is required, which permits the expansion of services to a sufficient degree for each customer. This implies that the important advantage of scalability is based on such adequacy of memory in all species. The same applies to the cache, the channels and broadband[9].

1.1 Thesis Contributions

This thesis relates to the judgment of the memory of cloud data centers with disaggregated (or disintegrated) servers, in other words servers whose ingredients or resources are in separate -concerning the physical location- subsystems[17, 32, 16, 23]. This type of servers has been under research lately, and the advantages and disadvantages are presented. The reason for using cloud computing is that it offers a better exploitation of the resources. In chapter 2 of this thesis, we will make extensive reference to the theoretical background of that architecture.

In this thesis the design and implementation of a complete Memory System Simulator is presented, as well as, the memory system evaluation of disaggregated cloud data centers.

The complete design of the Memory System Simulator, was carried out in co-operation with Orion Papadakis [26]. That includes the Instrumentation of a binary executable, its Analysis, the Generation of its Memory Trace, the Simula-

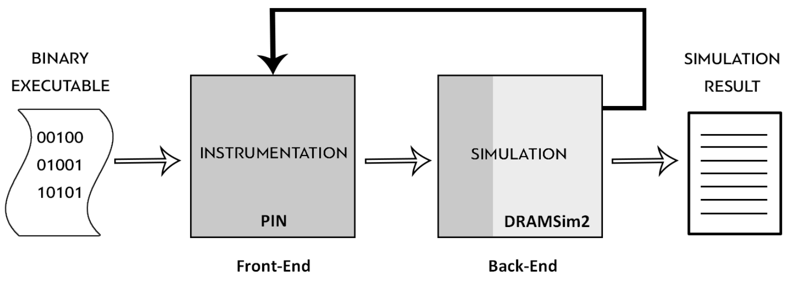


Figure 1.1: Whole System Abstract Figure.

tion of that Trace and the Exertion of Final Results. The Simulator was based on DRAMSim2[28], a Cycle Accurate Memory System Simulator for modelling and simulation, as the back-end, while the front-end driver was developed from scratch.

The current thesis focuses on the front-end driver of our Simulator. In order to implement that, we have created a tool, using Intel's PIN[27, 22], a dynamic binary instrumentation framework. Our PINtool simulates the functionality of all Cache and TLB levels (of a given CPU-model) and generates a Trace File with all RAM Accesses, including their Type of Access(Read/Write) and their Clock Cycle. That Trace File is necessary for the back-end of the Simulator, which was implemented by Orion Papadakis. To achieve the pairing between front-end and back-end, DRAMSim2 was included, as a library on our PINtool, in order to export the final `.log` file with the Simulation Results.

Our Simulator has been designed to be used on Disaggregated Cloud Servers, in order to, effectively, draw conclusions about their Memory Usage on different scenarios, by changing the Disaggregated/Local Memory Percentage.

1.2 Thesis Outline

In Chapter 2 we provide the theoretical background needed in order for the reader to comprehend the need of an efficient Cloud Computing Memory System and also the way to do the Instrumentation. All four forms of Instrumentation, and their combinations, are presented and so is the Intel PIN framework, the tool used for the Cloud Server's Instrumentation. Finally, we present the theoretical background of the Disaggregated Architectures. In Chapter 3 of this thesis, we present the related work done in the field of Memory Simulation, using Instrumentation tools, as well as, research about Disaggregated Cloud Data Centers. In Chapter 4 we present the complete Design of our Simulator, describing, in detail, all steps of it.

Consequently, Chapter 5 contains the results derived from the Simulations made by our tool. We have used Cloudsuite Benchmarks, for our Simulations, in order to evaluate the performance and efficiency of the cloud services to be assessed, also considering some different Disaggregated Architectures. Finally, Chapter 6 concludes the thesis and provides comments for possible future work. In Appendix A the reader will find some useful codes that refer to different points of the thesis.

CHAPTER 2

Theoretical Background

This thesis identifies three specific restrictive elements concerning Cloud Computing; three restrictions which are imported because of the way current data centers are built. These are [17]:

- The proportionality of the resources of the entire system follows that of the mainboard (basic building block). Thus, it is not possible to focus on certain elements that are crucial priorities, such as memory, so each system upgrades should involve all elements, as they existed in its first manufacturing and assembly.

- There is inadequate allocation of computing resources because of their concentration in one spot and their endorsement as a compact whole. The computational processing resources (CPU cores) can be fully exploited, while, conversely, utilizing only a relatively small amount of memory. Concentration of the cloud server components in a system thereby, leads to a fragmentation and great inefficiencies of their resources.

- Even when a single cloud server component is upgraded, the entire system of the server board is affected. Conversely, a disaggregated cloud server's component could, with great discretion, be upgraded.

In order to solve these problems, the disaggregation or disintegration (fragmentation) of cloud servers was invented[17, 20, 21, 32]. However, as mentioned above, there are some critical elements of cloud computing necessary to operate with completeness in the above objectives. The prerequisite to do this is the installation and operation of specific control methods in the code source and binary form. One of these ways will be analyzed in all its forms in this chapter. Specifically, four of the main types of "*orchestration code*" as it is often called, will be referred to. These are binary, source, dynamic and static instrumentation code.

2.1 Instrumentation

The instrumentation is a specific term used to denote the process and technique of use and adding a control code that interferes with an intercalary way in a program or the programming environment, in order to monitor or modify any program behavior[8]. This technique proves very useful nowadays, since there are a number of problems associated with the development of a code, which can, then, be identified and eliminated. Thus, some of the objectives of the instrumentation code can be:

- Gathering metrics code values.
- Automated debugging code.
- Detection (of other) error code.
- Memory Leak Detection.

The general form of the process of instrumentation of the code also includes some basic steps applicable to all sub. These are:

1. Determination of the points where the instrumentation will be implemented.
2. Introduction and implementation of specific form of the instrumentation code.
3. Taking up the control of the program (compiler / linker, etc.).
4. Saving the program execution parameters.
5. Execution of the Control Code (instrumentation).
6. Restoration of the pre-existing program execution parameters.
7. Returning the control to the main program.

2.1.1 Source Instrumentation

The first type of instrumentation code to which we will refer to, is the source code instrumentation[11]. The special feature of this format is that instrumentation applies to the source code, which is written and performed at the level used in this codes programming language.

2.1.2 Binary Instrumentation

Binary (or bytecode) instrumentation has to do with the binary code[2]. That is, its basic difference, regarding the other types of instrumentation and especially source instrumentation, is that it refers to the microprogramming/assembly form of the code, somewhere between the source and the machine code and, in fact, one level above the latter. The binary instrumentation which is often used, is the dynamic binary instrumentation, but there is the static one as well[5]. The diagram below depicts a scheme pertaining to the dynamic type (DBI) (Figure 2.3).

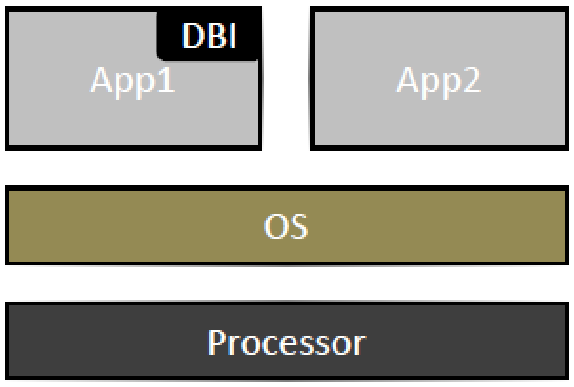
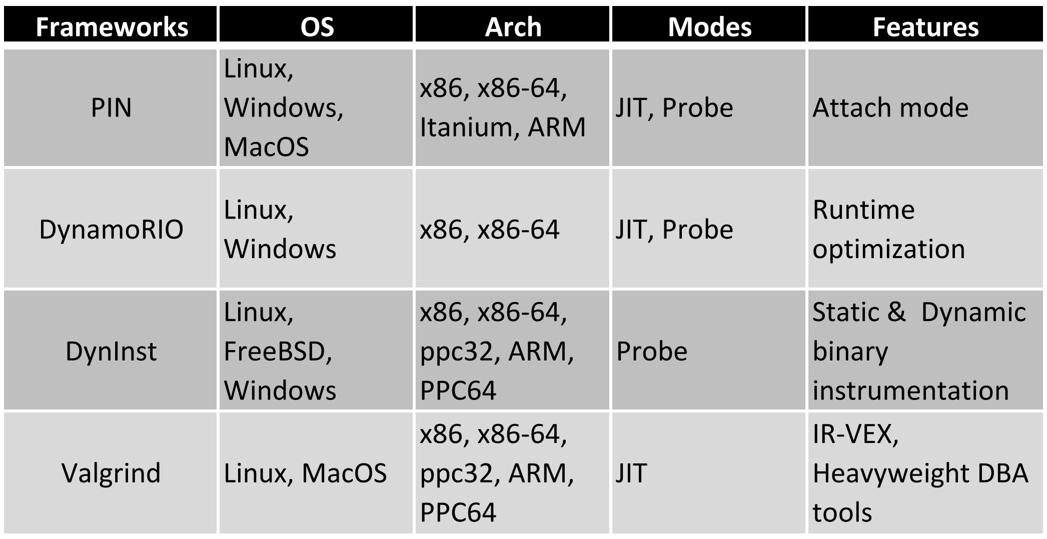


Figure 2.1: This diagram depicts a scheme pertaining to the dynamic type (DBI).

We will refer to DBI, as well as to the additional aforementioned terms, in the subsequent paragraphs.

2.1.3 Dynamic Instrumentation

Dynamic instrumentation is a more specific technique which is an important and very frequently used element within the broader framework of instrumentation techniques. Binary instrumentation is the most usual context of binary instrumentation, thus formulating dynamic binary instrumentation or Dynamic Binary Instrumentation (DBI). The most distinctive characteristic of DBI is that the instrumentation process takes place shortly before it is executed. As a consequence, recompilation or re-linking of the code is not indispensable, as in the static one, as will be seen in the forthcoming paragraph. This concept is named as Just In Time or JIT[5] . The code is actually attached to processes already being executed, hence the DBI code is considered to be injected into the whole code corpus. DBI is implemented by tools such as PIN[27, 22], DynamoRIO[3, 30], DynInst[29] and Valgrind[24].



Frameworks	OS	Arch	Modes	Features
PIN	Linux, Windows, MacOS	x86, x86-64, Itanium, ARM	JIT, Probe	Attach mode
DynamoRIO	Linux, Windows	x86, x86-64	JIT, Probe	Runtime optimization
DynInst	Linux, FreeBSD, Windows	x86, x86-64, ppc32, ARM, PPC64	Probe	Static & Dynamic binary instrumentation
Valgrind	Linux, MacOS	x86, x86-64, ppc32, ARM, PPC64	JIT	IR-VEX, Heavyweight DBA tools

Figure 2.2: DBI Frameworks.

There are two kinds of DBI tools in all:

- **Instrumentation code parts.** The instrumentation parts are injected just before a specific instruction which exists for the first time and defines the place that instrumentation is implemented.

- **Analysis code parts.** The analysis code parts are inserted each time an instruction is met and define the specific instrumentation functionalities which are to be implemented in the code.

Pros & Cons: start-up times are much slower and the demand of memory usage is increased as each method body has to be reallocated. The same applies to the CPU load to instrument on the fly.

Dynamic analysis implies capability to administer code which is generated in a dynamic or self-modifying manner.

2.1.4 Static Instrumentation

Static instrumentation appears, as the dynamic one, in the context of binary instrumentation. Thus, static binary instrumentation has got certain attributes according to which it functions. The basic elements which differentiate it from the dynamic type is that there are additional parts of code which are included beforehand (not "Just In Time"), namely a segment and data. An other difference is that the instrumentation occurs before even the program starts running. Furthermore, the header may be edited, as the following image indicates. The important thing is that the probes are baked inside the binary. This can be done in two ways:

First by having the compiler put the probes in.[15] Second, by what is called post link instrumentation. This gives immediate access to all the methods and instruments included in the binary and makes all the necessary changes.

Pros & Cons: This approach accomplishes much faster start-up times, since the images are loaded in to memory are the image that run.

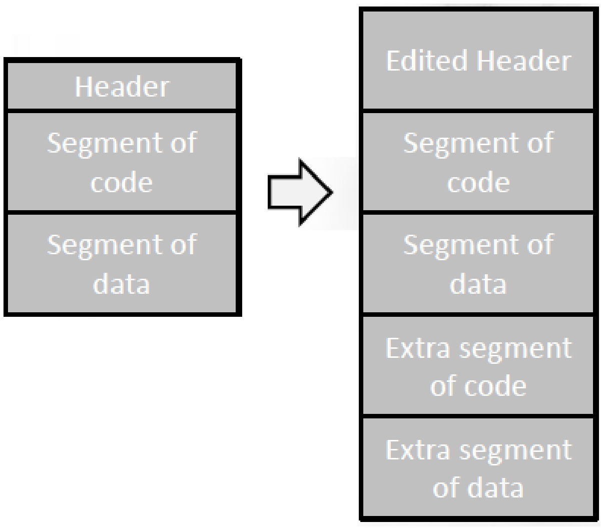


Figure 2.3: This diagram depicts a scheme pertaining to the dynamic type (DBI).

2.2 Intel PIN : A Dynamic Binary Instrumentation Tool

Pin is a framework for dynamic binary instrumentation[27, 22]. It supports the Android, Linux, OS X and Windows operating systems and executables for the IA-32, Intel(R) 64 and Intel(R) Many Integrated Core architectures.

The functionality of this framework is the insertion of an arbitrary code (written in C or C++) in arbitrary places in the executable. While the executable is running, the code is added dynamically, something that also makes the attachment of the Pin to an already running process, possible.

"*Pin provides a rich API that abstracts away the underlying instruction set idiosyncrasies and allows context information such as register contents to be passed to the injected code as parameters. Pin automatically saves and restores the registers that are overwritten by the injected code so the application continues to work. Limited access to symbol and debug information is available as well.*" as described

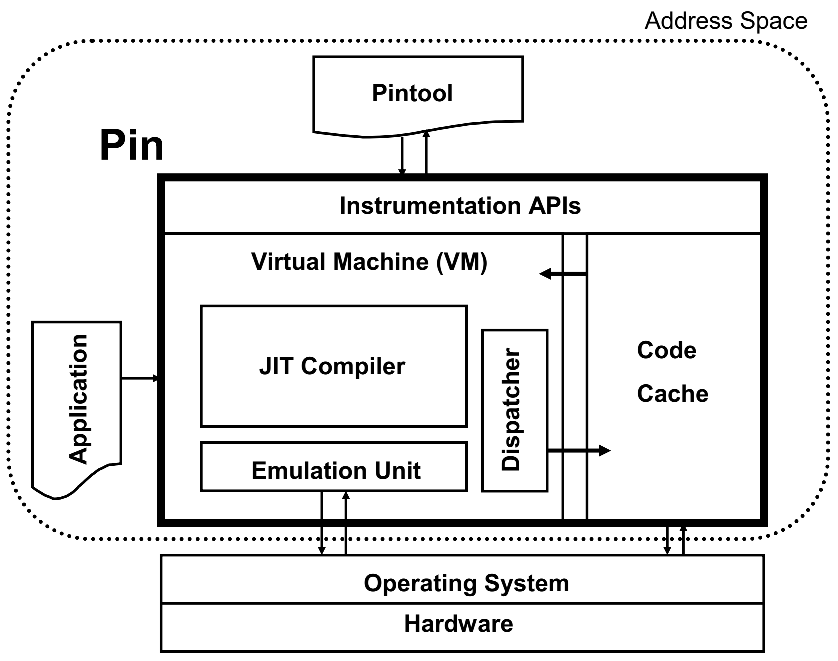


Figure 2.4: The architecture of Pin.

in the Intel PIN's user's manual. It is a proprietary software. However it is freely available for persons or entities using it to non-commercial ends.

Some of the advantages of Pin Instrumentation, as they have, already, been described, are the following:

- **Easy-to-use Instrumentation**; Uses dynamic instrumentation. (Does not need source code, recompilation, post-linking)
- **Programmable Instrumentation**; Provides rich APIs to write in C/C++ your own instrumentation tools.
- **Multiplatform**; Supports x86, x86-64, Itanium, Xscale. Supports Linux, Windows, MacOS

- **Robust**; Instruments real-life applications: Database, web browsers, Instruments multithreaded applications, Supports signals
- **Efficient**; Applies compiler optimizations on instrumentation code

Pin includes the source code for a large number of example instrumentation tools, like basic block profilers, cache simulators, instruction trace generators, etc. It is easy to derive new tools using the examples as a template.

The PIN engine uses the so-called PIN tools, for example an Instruction Counting Tool. The two main components of a Pin tool are: **instrumentation** and **analysis** routines:

- The *Instrumentation routine's* role is to insert calls to user defined analysis routines, by utilizing the rich API provided by Pin. These calls are inserted at arbitrary points in the application instruction stream. The basic characteristics of an application to instrument, are defined by the instrumentation routines.
- *Analysis routines* are called by the instrumentation routines at application run time.

For example, a user can write an instrumentation routine that instruments every instruction executed by an application via the Pin API [4]. This Pintool can count the total number of dynamic instructions executed by the program, once the instrumentation routine sets up a call to the user-defined analysis routine DoCount() (which increments a counter)

Moreover, Intel's PIN provides further advanced features, helpful in a variety of microarchitecture studies.

For example, pintools can:

- profile the dynamic or static distribution of instructions executed by a given application.
- acquire effective addresses of all memory instructions executed.
- determine the outcomes of branch instructions and their associated branch targets.
- change architectural state of registers.

All the above information provides users with customizable Pintools which model branch predictors, simple performance models and cache simulators.

Some of the basic arguments through which it is determined how instrumentation is carried out, are the following:

- `IPOINT_BEFORE`: a call is inserted before an instruction or a routine.
- `IPOINT_AFTER`: a call is inserted at the final paths of an instruction or a routine.
- `IPOINT_ANYWHERE`: an instrumentation call is made anywhere within the additional code corpus(that is, in a trace the PIN callback-function or a bbl).
- `IPOINT_TAKEN_BRANCH`: an instrumentation call is inserted on the taken edge of a branch, given that all the proper alterations are made.

An example code which includes such an argument is the following:

```
void Instruction(INS ins, void *v)
{
  INS_InsertCall(ins, IPOINT_BEFORE,
  (AFUNPTR)docount, IARG_END);
}
```

Listing 2.1: An example of Instruction() function

Let it be noted here, that the PIN is initiated through the use of the following instruction: `PIN_Init(argc, argv)`; Subsequently, in order to perform the instrumentation for the next instruction encountered, PIN uses the elements analyzed below:

1. The Function by the name `Instruction()`.
2. The function `INS_InsertCall()`. This function takes, among others, the arguments' values specified above, as acceptable values for its second argument.
3. The function `Fini()`. This function is called when the application is about to exit the program .

Instrumentation in PIN can be done at three different granularities[27, 22]:

1. Instruction
2. Basic block
   - A sequence of instructions terminated at a control-flow changing instruction
   - Single entry, single exit

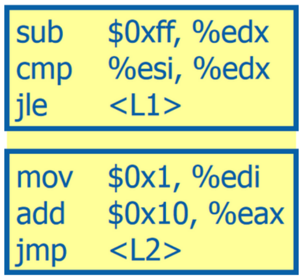


Figure 2.5: 1 Trace, 2 BBLs, 6 instructions

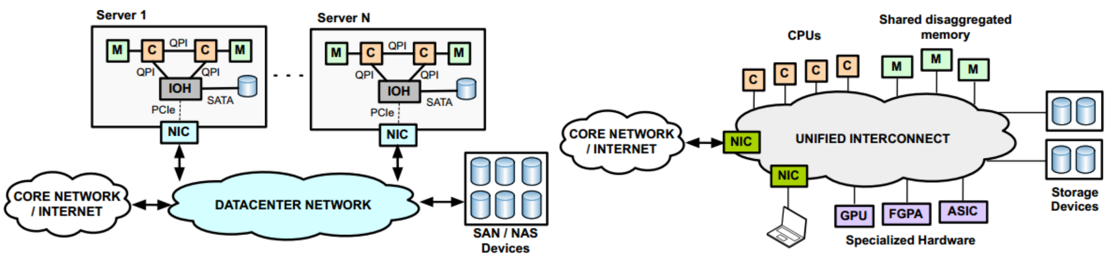
3. Trace

   - A sequence of basic blocks terminated at an unconditional control-flow changing instruction
   - Single entry, multiple exits

Every PIN tool performs different tasks to the instructions of the instrumented program, and for every task there is a different function which implements it. For example, the functions `docount()` or `printip()` are utilized, in order to perform instruction counting or related IP printing. The same rules apply to the rest of the PIN tools of the PIN engine.

2.3 Disaggregated Architecture

Disaggregated architecture is, as stated earlier, an answer provided to the challenges of our times concerning computing in general and cloud computing in specific, as well as drawbacks which are presented by systems developed nowadays. These drawbacks and challenges are enumerated and presented in a comprehensive way by Katrinis et al.)[17]



(a) Current datacenter (b) Disaggregated datacenter

Figure 2.6: Architectural differences between server-centric and resource-centric datacenters.

2.3.1 Disaggregated System Design

Disaggregation, by definition, means “separation into components”. Disaggregated (or disintegrated) systems are systems which have components spread across several physical locations. These systems are in a position to address certain issues of paramount importance, such as scalability and (modular) heterogeneity[12]. This type of architecture seems to introduce certain new challenges, such as, the need to diminish a new overhead which appears, due to new forms of latency, which show up in the architecture. However, it is the advantages of this architecture that outweigh its disadvantages. One of the main goals of disintegrated architectures is to promote and elevate the utilization degree of each component of a server or data center. Such components are the ones pertaining to memory, (central, graphical etc.) processing units, acceleration and storage units. It has been noted that due to the integration of all these components into concrete computing units and servers, there can be a massive -larger than 50%- underutilization. Every upgrade, on the other hand, has to see these systems as a whole. Disintegration reverses this process, which ceases to exist; it offers a much greater degree of flexibility, so that the redundancy of systems hardware and the associated costs are significantly lowered in a cost-effective way.

Especially, it is the memory quantity, quality and cost that have, long, been a difficult problem to process and solve. As described above, it is especially the cloud computing applications which are memory-intensive, thus they require a more centered approach. It seems that the disintegrated architecture is able to tackle this issue in a very efficient way. The issues solved[12] are the ones described above, through the separate management and upgrade of each component.

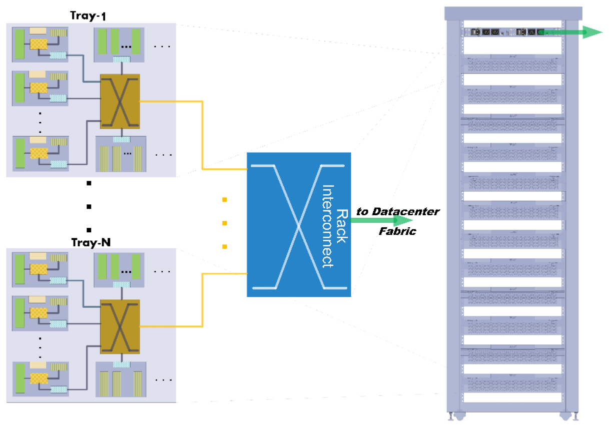


Figure 2.7: Block Diagram of high-level dReDBox Rack-scale architecture

Technologies used here are buffers-on-board, hierarchical buffers and disintegrated, master and slave controllers. All these innovative technologies contribute, as a whole, to the complete and efficient partitioning of the memory controllers. It is notable that this string of ideas was taken into careful consideration by the developers of an innovative project called dReDBox[17], which was funded by the EC (European Commission). This project seems to place a great emphasis on the virtual memory management aspect of the disintegration process, as it implements physical and logical attachments of remote memory. This emphasis is explained by what has been mentioned in this, as well as previous paragraphs of this thesis.

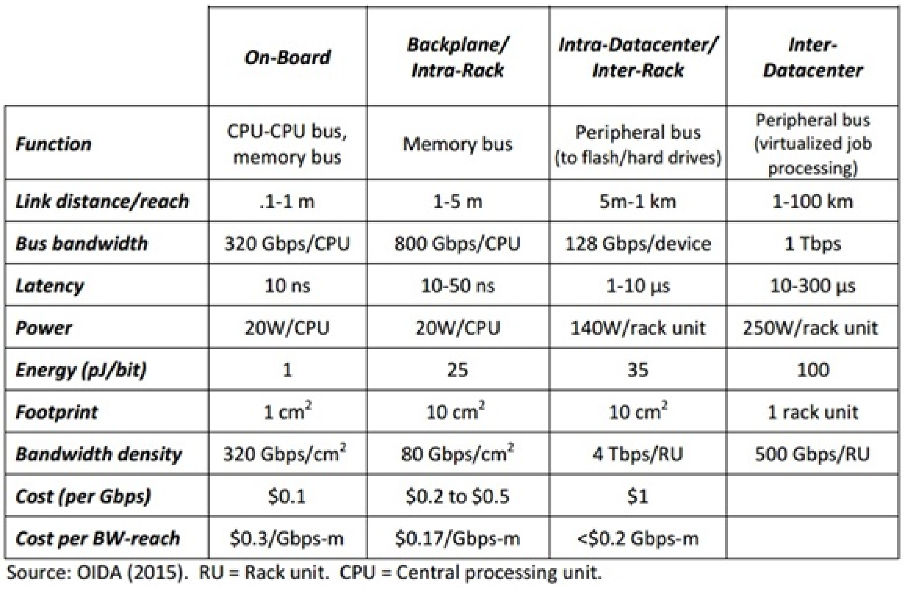
2.3.2 Replace electrons with photons

Though disaggregation has many pros, we must keep in mind that using discrete resource pools means developing interconnections previously included on the server motherboard. Beyond doubt, PC-board electrical interconnections have almost excelled physically and electronically.

The answer may lie in the science of photonics. Photonics are defined as "a research field whose goal is to use light to perform functions that traditionally fell within the typical domain of electronics such as telecommunications and information processing."

Photons are already used to disseminate traffic from data center racks to the rest of the digital world. Researchers are finding ways to replace electrical signals all the way to the processor silicon, which raises the question of the use of fiber-optic runs to interconnect the resource pools.

The members of OSA (The Optical Society)Industry Development Associates and the Center for Integrated Access Networks when the organizations hosted a workshop focused on how photonics may facilitate data-center disaggregation[7]. The workshop also attempted to find the way optics can enable a migration to disaggregation, describe the appearance the photonics-enabled disaggregated data



	On-Board	***Backplane/ Intra-Rack***	***Intra-Datacenter/ Inter-Rack***	***Inter-Datacenter***
Function	CPU-CPU bus, memory bus	Memory bus	Peripheral bus (to flash/hard drives)	Peripheral bus (virtualized job processing)
Link distance/reach	.1-1 m	1-5 m	5m-1 km	1-100 km
Bus bandwidth	320 Gbps/CPU	800 Gbps/CPU	128 Gbps/device	1 Tbps
Latency	10 ns	10-50 ns	1-10 μs	10-300 μs
Power	20W/CPU	20W/CPU	140W/rack unit	250W/rack unit
Energy (pJ/bit)	1	25	35	100
Footprint	1 cm^2	10 cm^2	10 cm^2	1 rack unit
Bandwidth density	320 Gbps/cm^2	80 Gbps/cm^2	4 Tbps/RU	500 Gbps/RU
Cost (per Gbps)	$0.1	$0.2 to $0.5	$1	
Cost per BW-reach	$0.3/Gbps-m	$0.17/Gbps-m	<$0.2 Gbps-m	

Source: OIDA (2015). RU = Rack unit. CPU = Central processing unit.

Figure 2.8: Disaggregated data center interconnect requirements (Courtesy of The Optical Society) [7]

center and the performance metrics/requirements essential to photonics in disaggregated data centers

Disaggregation offers improved efficiency and increased capacity, but the benefits are limited by the cost and performance of the photonic interconnections. Latency requirements, in particular, impose hard limits on distances over which certain resources can be disaggregated.

The workshop attendees were optimistic about fiber-optic interconnects replacing electrical cables everywhere except on the rack shelf. Optics will out compete electrical interconnects for links that enable disaggregation if optics can hit the metrics of cost, performance, and size. The above table (Figure 2.8) lists the aforementioned metrics.

CHAPTER 3

Related Work

This thesis is related to projects which span several different topics, from instrumentation implementation to disaggregated (disintegrated) cloud servers discourses. Regarding the programming work which was carried out in this thesis, there are a lot of similarities with certain projects. We will refer to the most important one below, which, in a way, inspired the whole thesis.

This thesis is based on the PIN project, released by Intel in 2012[27, 22] . PIN uses, as has been mentioned above, instrumentation and analysis routines and is very popular, as well as very broadly used in scientific studies and publications(40.000 users, 400 scientific citations). The PIN project is the one from which this dissertation has derived a great deal of elements, as well as its fundamental notions and inspiration.

The work named "*Ramulator: A Fast and Extensible DRAM Simulator*"[18] is one similar to ours, fundamentally in terms of its DRAM usage and simulator. The authors contribute their own version of a DRAM simulator, which emphasizes strongly on the concept of extensibility. Ramulator is based on a modular design, thus promoting scalability, and supports several DRAM standards, such as DDR3/4 and LPDDR3/4. Ramulator also uses a hierarchical design of nodes (state machines). The authors conclude that such a simulator could facilitate memory-related research.

Finally, regarding the disaggregated (disintegrated) server or data-center part, we will refer here to a similar research, conducted by Han et al[12]. The authors examine disaggregated data centers, where resources such as memory, storage and communication resources are built up as separate groups of resources of the same kind. This type of data centers brings about, as the authors deduce, not only a greater level of modularity but an improved efficiency and performance, as well. Nevertheless, they stress that a make-or-break factor will be the network which, due to its structure, will have to be the host of the new disaggregated servers.

Many research efforts have focused on disaggregated memory with the aim to enable scaling of memory and processing resources at independent growth paces. Lim *et al.*[20, 21] presents the “memory blade” as an architectural approach to introduce flexibility in memory capacity expansion for an ensemble of blade servers. The authors explore memory-swapped and block-access remote access solutions and address software- and system-level implications by developing a software-based disaggregated memory prototype based on the Xen hypervisor. They find that mechanisms which minimize the hypervisor overhead are preferred in order to achieve low-latency remote memory accesses.

The dReDBox (disaggregated Recursive Datacenter in a Box)[17, 32, 16, 23] project tries to address the problem of fixed resource proportionality in next-generation, low-power data centers by proposing a paradigm shift toward finer resource allocation granularity, where the unit is the function block rather than the mainboard tray. This introduces various challenges at the system design level, requiring elastic hardware architectures, efficient software support and management, and programmable interconnect. Hardware accelerators can be dynamically assigned to processing units to boost application performance, while high-speed, low-latency electrical and optical interconnect is a prerequisite for realizing the concept of data center disaggregation.

4 CHAPTER

Implementation of DiMEM Simulator

4.1 Front-End of DiMEM Simulator: Instrumentation stage

DiMEM SIMULATOR

Figure 4.1: The DiMEM Simulator Logo.

The goal of this work is to simulate and evaluate a memory system of a Cloud Server. To achieve that, it was important to create the front-end driver of the DiMEM Simulator. This thesis focuses on the Functional Simulation of TLB and Cache hierarchy, as well as its use and exploitation for Main Memory Accesses Trace File generation. We have used an example Pintool for Memory Tracing, called *Allcache*[22]. Allcache is an ISA-portable PIN tool for functional simulation of instruction/data TLB and cache hierarchy. We have used it as a basic core to build our tool on it.

4.1.1 Functional simulation of TLB and cache hierarchy using PIN

First of all, we had to describe the TLB (Instruction TLB *&* Data TLB) and Cache hierarchy (L1, L2 etc, Private or Unified) in our PINtool. There are many different cache hierarchies, that depend on different architectures. On chapter 5, we will make extensive reference to the architectures we have chosen to simulate.

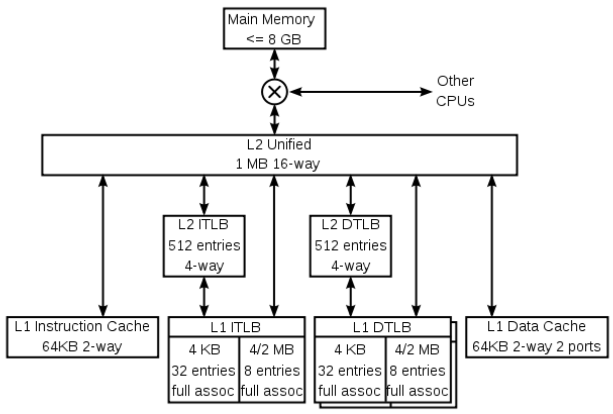


Figure 4.2: Cache hierarchy of the K8 core in the AMD Athlon 64 CPU.

4.1.1.1 Cache hierarchy

All memory references generated by a workload have to be captured and played through a cache model. By doing this the cache behavior of an application is simulated. To design the cache hierarchy in our tool, we have used a namespace for each cache level or TLB. In those namespaces we created a CACHE class, as defined on `pin_cache.H` header file (see Appendix A). On that class we have defined some necessary features of the Cache level. Those features are:

- **Cache Sets** such as direct-mapped, FIFO replacement (round-robin), least-recently used (LRU) for Read Misses.
- **Cache Allocation** such as write-allocate or write-no-allocate for Write Misses.

- **Cache Base** such as cache size, line size, associativity, max sets and max associativity.

The following code is an example of a Cache namespace:

```
namespace ITLB
{
// instruction TLB: 4 kB pages, 32 entries, fully associative
const UINT32 lineSize = 4*KILO;
const UINT32 cacheSize = 32 * lineSize;
const UINT32 associativity = 32;
const CACHE_ALLOC::STORE_ALLOCATION allocation = CACHE_ALLOC::
    STORE_ALLOCATE;
const UINT32 max_sets = cacheSize / (lineSize * associativity);
const UINT32 max_associativity = associativity;

typedef CACHE_ROUND_ROBIN(max_sets, max_associativity, allocation)
    CACHE;
}
LOCALFUN ITLB::CACHE itlb("ITLB", ITLB::cacheSize, ITLB::lineSize,
    ITLB::associativity);
```

Listing 4.1: *namespace ITLB, with 4 kB pages, 32 entries, fully associative*

4.1.1.2 Cache hierarchy for multicore

The future of high-performance and Cloud computing - as shown by the latest tendencies- will be defined by the performance of multi-core processors [1, 2, 3]; this resulting, in processor architects currently facing key design decisions in designing the memory hierarchy[13].

A Cache hierarchy, according to a modern multicore CPU, must follow the multicore structure's laws. To accomplish the actual parallel functionality, a processor's cache can not be completely shared, consequently, the existence of a private structure per core is, also, necessary. We can see an example of a Multicore cache hierarchy on Figure 4.3.

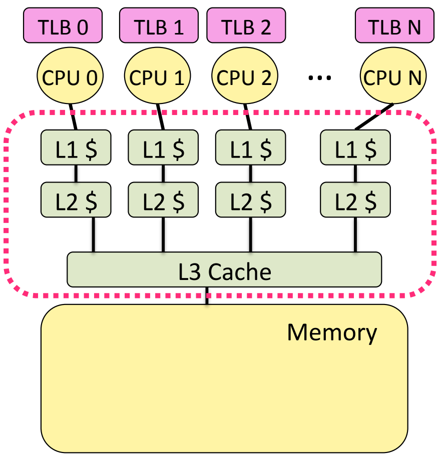


Figure 4.3: An example of a Multicore cache hierarchy

Allcache Pintool provides a single-core hierarchy example, as described above. In order to extend its functionality, for more than one cores, we had to render the Listing 4.1, as well as, all cache hierarchy namespaces, configurable. To accomplish that we had to insert a new Libraries package, in our code, the Boost C++ Libraries *(boost.org)*. Those libraries are based on the standard libraries and they are used to promote the flexibility and efficiency of code [6]. We chose to use the local.hpp library, which has been made to enable us use MACROs in C++.

We need MACROs to make a configurable cache hierarchy for every core of the system. To do that, we had to change the following code:

```
LOCALFUN ITLB::CACHE itlb("ITLB", ITLB::cacheSize, ITLB::lineSize,
    ITLB::associativity);
```

Listing 4.2: *ITLB::CACHE itlb()* for single-core

to make use of BOOST Libraries, as below:

```
extern ITLB::CACHE itlbs[CORE_NUM];

#define BOOST_PP_LOCAL_LIMITS      (0, CORE_NUM - 1)
#define BOOST_PP_LOCAL_MACRO(n)    \
ITLB::CACHE( "ITLB " #n, ITLB::cacheSize, ITLB::lineSize, ITLB::
    associativity),
```

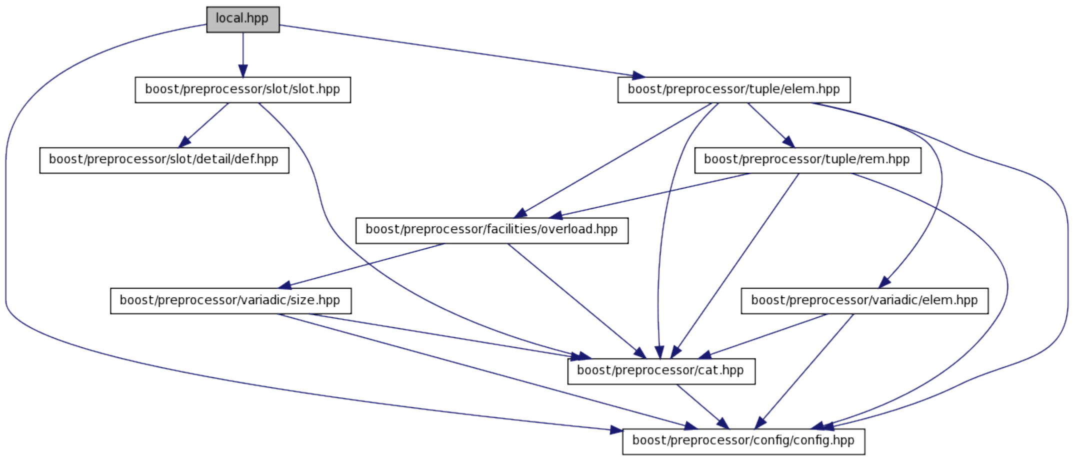

local.hpp
boost/preprocessor/slot/slot.hpp
boost/preprocessor/tuple/elem.hpp
boost/preprocessor/slot/detail/def.hpp
boost/preprocessor/tuple/rem.hpp
boost/preprocessor/facilities/overload.hpp
boost/preprocessor/variadic/size.hpp
boost/preprocessor/variadic/elem.hpp
boost/preprocessor/cat.hpp
boost/preprocessor/config/config.hpp

Figure 4.4: local.hpp File Reference

```
6
7  ITLB::CACHE itlbs[] =
8  {
9  #include "../../../boost_1_60_0/boost/preprocessor/iteration/detail/
       local.hpp"
10 };
```

Listing 4.3: *ITLB::CACHE itlb()* for multicore

We, finally, had to define the number of cores in our code, to make it fully customizable. So we also added in our code the following line:

```
#define CORE_NUM 4
```

Listing 4.4: Define the number of cores

`CORE_NUM` not only is an argument that defines the number of Private Cache levels (see lines 1 & 3 of Listing 4.3), but it also serves in many other points of our pintool.

In general, the modern CPUs have, at least, one Unified/Shared Cache level following the Private ones. That is not a commitment and it depends on each architecture (e.g. ARM Cortex A-53 does not have any Unified Cache level).

4.1.2 Instrumentation Function

The Instrumentation Function of our pintool is the `Instruction()` call back function. It is defined as Instrumentation Function of the pintool, in the `main()` function making use of `INS_AddInstrumentFunction(Instruction, 0)` API call.

```
LOCALFUN VOID Instruction(INS ins, VOID *v)
{
    if (INS_IsMemoryRead(ins) && INS_IsStandardMemop(ins))
    {
        const UINT32 size = INS_MemoryReadSize(ins);
        const AFUNPTR countFun = (size <= 4 ? (AFUNPTR) MemRefSingle
    : (AFUNPTR) MemRefMulti);

        // only predicated-on memory instructions access D-cache
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, countFun,
            IARG_THREAD_ID,
            IARG_MEMORYREAD_EA,
            IARG_MEMORYREAD_SIZE,
            IARG_UINT32, CACHE_BASE::ACCESS_TYPE_LOAD,
            IARG_UINT32, 'R',
            IARG_END);
    }

    if (INS_IsMemoryWrite(ins) && INS_IsStandardMemop(ins))
    {
        const UINT32 size = INS_MemoryWriteSize(ins);
```

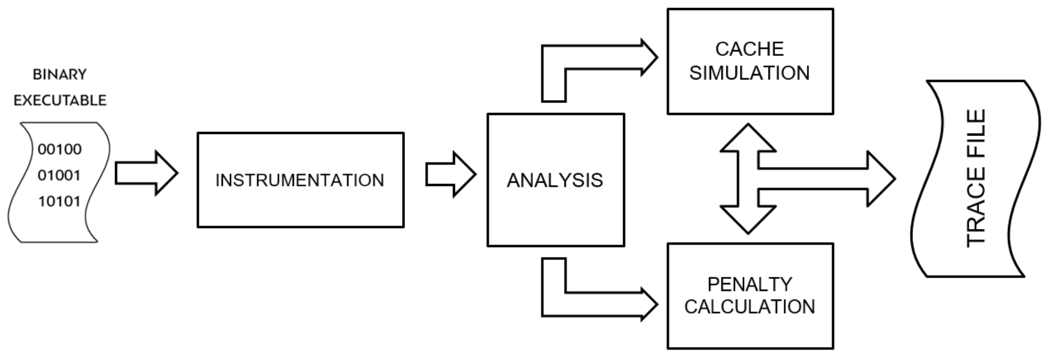


Figure 4.5: Front-End Diagram

```
        const AFUNPTR countFun = (size <= 4 ? (AFUNPTR) MemRefSingle
    : (AFUNPTR) MemRefMulti);

        // only predicated-on memory instructions access D-cache
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, countFun,
            IARG_THREAD_ID,
            IARG_MEMORYWRITE_EA,
            IARG_MEMORYWRITE_SIZE,
            IARG_UINT32, CACHE_BASE::ACCESS_TYPE_STORE,
            IARG_UINT32, 'W',
            IARG_END);
    }

}
```

Listing 4.5: Instrumentation Function

The Instrumentation Granularity we have used in our pintool, is the Instruction one, as we can see on the header of the Instrumentation Function (Listing 4.5). The functionality of Instrumentation Function is to check the memory operation existence (Read/Write), instruction following instruction. In case of a predicated Memory Read, it calls the suitable Analysis Function (before the instruction) passing through the following arguments: `IARG_THREAD_ID`, `IARG_MEMORYREAD_EA`, `IARG_MEMORYREAD_SIZE`, `CACHE_BASE::ACCESS_TYPE_LOAD`, `'R'` (access type flag). In case of a predicated Memory Write, the arguments are the following: `IARG_THREAD_ID`, `IARG_MEMORYWRITE_EA`, `IARG_MEMORYWRITE_SIZE`, `CACHE_BASE::ACCESS_TYPE_STORE`, `'W'` (access type flag). Our Pintool uses two Analysis Functions (`MemRefSingle()`, `MemRefMulti()`), as we can see above. Each one is appropriate for single or multiple memory operands instructions. `INS_MemoryReadSize(ins)` and `INS_MemoryWriteSize(ins)` are used to determine the total Memory size, that instruction reads or writes. If the total Memory size is less than or equal to 4 bytes, the `MemRefSingle()` Analysis Function is chosen. Otherwise the `MemRefMulti()` Analysis Function is chosen.

4.1.3 Analysis Functions

`MemRefSingle()` and `MemRefMulti()` functions are the "Alpha and Omega" of our pintool. Their main functionality is to distinguish the Cache Misses from Cache Hits, in order to collect only the Main Memory Read/Write accesses. A Main Memory access occurs when the required data can not be found in Cache (last-level Cache Miss).

The Analysis Functions operate by checking from the upper cache level (L1) to the lower ones (L2, L3...), until they reach a cache hit or finally a last-level cache miss.

```
LOCALFUN UINT32 MemRefSingle(UINT32 threadid, ADDRINT addr, UINT32
    size, CACHE_BASE::ACCESS_TYPE accessType, CHAR r)
{
        BOOL Miss = false;
        BOOL dl2Hit = false;
        // DTLB
        const BOOL dtlbHit = dtlbs.AccessSingleLine(addr, CACHE_BASE
    ::ACCESS_TYPE_LOAD);
        // first level D-cache
```

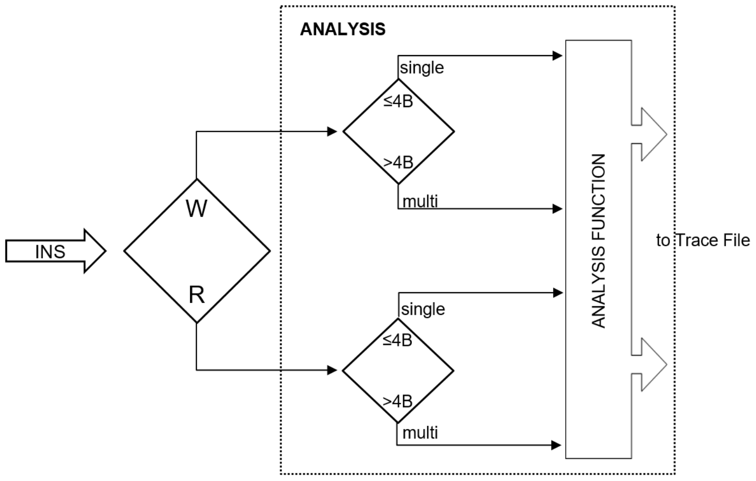


Figure 4.6: Analysis Function

```
 8        const BOOL dl1Hit = dl1s.AccessSingleLine(addr, accessType);
 9        // second level unified Cache
10        if ( ! dl1Hit)
11        {
12            dl2Hit = dl2s.Access(addr, size, accessType);
13            if ( ! dl2Hit)
14                 Miss = true;
15        }
16        penalty = PenaltyCalc(dtlbHit, dl1Hit, dl2Hit);
17        if (Miss)
18        {
19            RecordAtTable(threadid, addr, r, penalty, 'R');
20            ram_count++;
21        }
22        else
23        {
24            RecordCachePenalty(threadid, penalty);
25        }
26    return 0;
27 }
```

Listing 4.6: An Analysis Function without multithreading functionality

In case of a last level miss, the `RecordAtTable()` function is called (see line 19 on Listing 4.6). That function stores an instance of the Main Memory Access Record (MMAR) structure, (`Record()` in our pintool), into the Record Buffer (`recs()` in our pintool). So, the Record Buffer (RB) is a vector that contains the MMAR instances. Furthermore, the MMAR structure models a main memory access, with the following fields: Thread id, Instruction Address , Read/Write flag, Cache Penalty, Cycle, Cache/RAM flag. An MMAR instance is created into the `RecordAtTable()` function, which uses its attributes to set the struct fields. Some MMAR fields (threadId, ip, r, cacheflags) are set directly from the corresponding function fields, unlike the penalty field which has to be calculated.

```
static VOID RecordAtTable(UINT32 threadid, ADDRINT ip, CHAR r, UINT32
     penalty, CHAR cacheFlag)
{  //record at Buffer
    recs[threadid % CORE_NUM].push_back(Record());
    recs[threadid % CORE_NUM][j[threadid % CORE_NUM]].threadid=
    threadid;
```

```
 5    recs[threadid % CORE_NUM][j[threadid % CORE_NUM]].ip=ip;
 6    recs[threadid % CORE_NUM][j[threadid % CORE_NUM]].r=r;
 7    recs[threadid % CORE_NUM][j[threadid % CORE_NUM]].penalty=penalty
      + cachePenalties[threadid % coreNUM];
 8    recs[threadid % CORE_NUM][j[threadid % CORE_NUM]].cycle=0;
 9    recs[threadid % CORE_NUM][j[threadid % CORE_NUM]].cacheFlag=
       cacheFlag;
10    j[threadid % CORE_NUM]++;
11    cachePenalties[threadid % CORE_NUM] = 0;
12 }
```

Listing 4.7: RecordAtTable() with multithreading functionality

Between two Main Memory Accesses (`MMAR`$_1$, `MMAR`$_2$) which are stored in the RB, there have, actually, been much more -not stored- Cache Hits. These are useless for the Simulation, but their additive cache penalty is equal to the Cycles between `MMAR`$_1$ and `MMAR`$_2$. This functionality has been achieved by inserting the `RecordCachePenalty(threadid, penalty)` in the Analysis Functions, to sum up each Cache Penalty, between two Cache Misses.

```
static VOID RecordCachePenalty(UINT32 threadid, UINT32 penalty)
{
     //keep the additive penalty
     cachePenalties[threadid % CORE_NUM]=cachePenalties[threadid %
    CORE_NUM] + penalty;
}
```

Listing 4.8: RecordCachePenalty() with multithreading functionality

The Record Buffer's Penalty is calculated by adding the additive cache penalty to the current penalty (see line 9 on Listing 4.7). The current Penalty is calculated by using the `PenaltyCalc(dtlbHit, dl1Hit, dl2Hit)` function.

```
LOCALFUN std::size_t PenaltyCalc (BOOL bool_tlbHit, BOOL bool_l1Hit,
    BOOL bool_l2Hit)
{
     std::size_t int_tlbHit = (bool_tlbHit) ? 1 : 0;
     std::size_t int_l1Hit = (bool_l1Hit) ? 1 : 0;
     std::size_t int_l2Hit = (bool_l2Hit) ? 1 : 0;
     return (int_tlbHit*TLB_PENALTY) + (int_l1Hit*L1_PENALTY) + (
     int_l2Hit*L2_PENALTY);
}
```

Listing 4.9: PenaltyCalc() function

`PenaltyCalc()` uses the following boolean attributes to calculate the Penalty of an Instruction: `dtlbHit, dl1Hit, dl2Hit, ...` in combination with the defined values of `TLB_PENALTY, L1_PENALTY, L2_PENALTY, ...` (see Listing 4.9).

```
#define TLB_PENALTY 20
#define L1_PENALTY 3
#define L2_PENALTY 15
```

Listing 4.10: Cache Level Latencies and DTLB Miss Penalty #defines

4.1.4 Instrumentation Output

As described above, the back-end driver of our Simulator is DRAMSim2[28]. Trace-driven simulation is a popular technique for conducting memory performance studies. In standalone mode, DRAMSim2 can simulate Memory-System traces. A suitable trace file for DRAMSim2 has the following format:

```
0x7f64768732d0 P_FETCH 1
0x7ffd16a5a538 P_MEM_WR 8
0x7f6476876a40 P_FETCH 12
0x7f6476a94e70 P_MEM_RD 61
0x7f6476a95000 P_MEM_RD 79
```

Listing 4.11: A DRAMSim2 Trace File sample

The first column indicates the Instruction's Virtual Address, the second one the Instruction's Type and the last column indicates the Cycle of each Instruction. To communicate the front-end driver, with the analogous one back-end, it is necessary for our pintool to generate the appropriate Trace Sequence.

4.1.4.1 Trace file

The functionality of writing into a Tracefile, has been added to our Simulator. To achieve that, the following Command-line Switches have been added:

```
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o",
    "TraceFile.trc", "specify trace file name");
KNOB<BOOL> KnobValues(KNOB_MODE_WRITEONCE, "pintool", "values", "1",
    "Output memory values reads and written");
```

as well as, the following lines in the `main()` function:

```
TraceFile.open(KnobOutputFile.Value().c_str());
TraceFile.setf(ios::showbase);
```

Now, we had to load the `Tracefile.trc` like the three-column DRAMSim2 Tracefile format:

```
for (i=0;i<rec.size();i++)
{
  if(rec[i].cacheFlag=='R')
  {
    if(rec[i].r=='F')
    {
      TraceFile << hex << rec[i].ip << " " << "P_FETCH" << " " << dec
    << rec[i].penalty;
            TraceFile << endl;
        }
        else if(rec[i].r=='W')
        {
            TraceFile << hex << rec[i].ip << " " << "P_MEM_WR" << " "
    << dec << rec[i].penalty;
            TraceFile << endl;
        }
        else
        {
            TraceFile << hex << rec[i].ip << " " << "P_MEM_RD" << " "
    << dec << rec[i].penalty;
            TraceFile << endl;
        }
     }
}
```

Listing 4.12: Trace File generation code

4.1.4.2 Record Buffer

In the shared-library mode, DRAMSim2 exposes the basic functionality by creating a new object (MemorySystem) and by adding requests to that. There is no need for an external Tracefile, in the shared-library mode. The role of the Tracefile has been played by the Record Buffer (RB), as described above. In fact, RB is the source of the Tracefile.

4.1.5 Multithreading Functionality

Pin supports the instrumentation of multi-threaded applications, along with the single-threaded ones. The operating system controls the scheduling of different threads of the application. Pin charges each thread with a unique ID different from the native process ID assigned by the operating system. It does so in order to distinguish between the different threads of the application. Pin assigns the 1st thread (i.e. the main thread) with thread ID 0 and each additional new thread the next sequential ID (i.e. 1, 2, 3) and so on. This way, when a five-threaded workload is used to conduct a study, PIN distinguishes between threads by assigning each thread a different ID, starting from ID 0 (for the main thread) and continuing with the remaining four threads.

The philosophy of multithreading in our Pintool is that every system core deals with more than one threads simultaneously. The multi-threading functionality has been achieved by making the following changes on our pintool. First of all, we had to change the Cache hierarchy code, as described on subsection 4.1.1.2 above. The next step was to add the `thread_data_t` class and the `ThreadStart()` function,

```
#define PADSIZE 56  // 64 byte line size: 64-8

// a running count of the instructions
class thread_data_t
{
  public:
  thread_data_t() : _count(0) {}
  UINT64 _count;
  UINT8 _pad[PADSIZE];
};

// key for accessing TLS storage in the threads. initialized once in
    main()
static  TLS_KEY tls_key;

// function to access thread-specific data
thread_data_t* get_tls(THREADID threadid)
{
  thread_data_t* tdata =
  static_cast<thread_data_t*>(PIN_GetThreadData(tls_key, threadid));
  return tdata;
}

VOID ThreadStart(THREADID threadid, CONTEXT *ctxt, INT32 flags, VOID
    *v)
{
  PIN_GetLock(&lock, threadid+1);
  numThreads++;
  PIN_ReleaseLock(&lock);

  thread_data_t* tdata = new thread_data_t;

  PIN_SetThreadData(tls_key, tdata, threadid);
}
```

Listing 4.13: Multithreading Functionality

in our Pintool. They give the access to a Pintool for dealing with more than one threads simultaneously, but they can, only, work by combining them with the functionality of Locks (which allow only one thread to enter the part that's locked and the lock is not shared with any other processes) and Mutexes (same as locks, but shared by multiple processes). To add Lock/Mutex on our Pintool, the following code has been added in the `main` function:

```
 // Initialize the lock
PIN_InitLock(&lock);

// Initialize the mutexes
PIN_MutexInit(&Mutex);

// Obtain  a key for TLS storage.
tls_key = PIN_CreateThreadDataKey(0);

// Register ThreadStart to be called when a thread starts.
PIN_AddThreadStartFunction(ThreadStart, 0);
```

Listing 4.14: Initialize the lock/mutex

as well, the following in the `Fini` function:

```
PIN_MutexFini(&Mutex);
```

Every time, now, the Pintool is dealing with a threadId, it is enclosed between a Lock or a Mutex. For example:

```
static VOID RecordAtTable(UINT32 threadid, ADDRINT ip, CHAR r, UINT32
     penalty, CHAR cacheFlag)
{    //record at Buffer
PIN_MutexLock(&Mutex);
recs[threadid % CORE_NUM].push_back(Record());
recs[threadid % CORE_NUM][j[threadid % CORE_NUM]].threadid=threadid;
recs[threadid % CORE_NUM][j[threadid % CORE_NUM]].ip=ip;
recs[threadid % CORE_NUM][j[threadid % CORE_NUM]].r=r;
recs[threadid % CORE_NUM][j[threadid % CORE_NUM]].penalty=penalty +
    cachePenalties[threadid % coreNUM];
recs[threadid % CORE_NUM][j[threadid % CORE_NUM]].cycle=0;
recs[threadid % CORE_NUM][j[threadid % CORE_NUM]].cacheFlag=cacheFlag
    ;
j[threadid % CORE_NUM]++;
cachePenalties[threadid % CORE_NUM] = 0;
PIN_MutexUnlock(&Mutex);
}
```

Listing 4.15: Mutex example

Listing 4.15 indicates the last (but not least) change of the multithreading functionality, we have rendered. Every one-dimensional table and vector had to be replaced by a multi-dimensional one, to "simulate" the multicore design.

I.e. the `recs.push_back(Record())` has been replaced by `recs[threadid %`

`coreNUM].push_back(Record())`, where `threadid % coreNUM` is the new functionality, which has been added for making the right Core choice. E.g. a five-threaded workload in a 2-core simulation, will be treated like that:

Table 4.1: Multithreading example

ThreadID	Core 0	Core 1
0	✓	
1		✓
2	✓	
3		✓
4	✓	

4.1.6 Hyperthreading Functionality

Hyperthreading technology allows a single physical processor core to behave like 2 (or more) logical processors[1]. The processor can run 2 (or more) independent applications at the same time. Hyperthreading technology allows a single processor core to execute 2 (or more) independent threads simultaneously

While hyperthreading does not double the performance of a system, it can increase performance by better utilizing idle resources leading to greater throughput for certain important workload types. An application running on one logical processor of a busy core can expect slightly more than half of the throughput that it obtains while running alone on a non-hyperthreaded processor. Hyperthreading performance improvements are highly application-dependent, and some applications might see performance degradation with hyperthreading because many processor resources (such as the cache) are shared between logical processors.

[1]To avoid confusion between logical and physical processors, Intel refers to a physical processor as a socket.

As is clear from the above, the Hyperthreading Functionality is essential to a modern processor. To add that functionality in our Pintool, the following additions, have been made:

```
#define HYPERTHREADING 2   // THREADS PER CORE, 1 = NO HYPERTHREADING

coreNUM = CORE_NUM*HYPERTHREADING;
```

as well as, the `CORE_NUM` has been replaced by `coreNUM`, everywhere except in the Cache Hierarchy part, in order to maintain the logical processors, as they were.

4.2 Back-End of DiMEM Simulator: Simulation stage

The Simulation Stage of the DiMEM Simulator was designed by Orion Papadakis[26]. His Diploma Thesis focuses on the Pintool's Simulation Preparation functionality, both from theoretical and implementation point of view, so that the Pintool obtains more complete system and useful tool characteristics than a simple Memory Tracing Pintool. The implemented features which have been added on the pintool are the following:

1. Reordering, Approximate Timing and Sorting of the Multithreaded Trace
2. Making use of the capabilities of DRAMSim2 Cycle Accurate Memory System Simulator,
3. Implementation of Disaggregate Memory Behaviour
4. Speeding-up the Simulation with “Skip Mode”.

CHAPTER 5

Evaluation and Experimental Results

5.1 Evaluation

The section of Evaluation is about the CPU, DRAM and Benchmark choices. The first are presented by CPU Metrics and the DRAM model are described in the DRAM Metrics. After that, a Cloudsuite benchmark suites summary performed.

5.1.1 CPU Metrics

As it has already been explained, the DiMEM Simulator is ISA portable. That feature gives the user the valuable freedom of selection, to decide which CPU Model he wants to use coupled with a Memory System. The user can modify the Cache Size, Associativity, Penalties, the number of Cores, to enable/disable the Hyperthreading etc.

Three popular CPUs for Cloud Servers, are used in the current thesis:

1. ARM Cortex-A53
2. ARM Cortex-A57
3. IBM Power8

Their metrics are presented below:

	ARM Cortex-A53	ARM Cortex-A57	IBM Power8
Cores	4	8	20
Hyperthreading	2 threads/core	2 threads/core	8 threads/core
Logical Cores	8	16	160
Cache	2-levels and TLB	3-levels and TLB	4-levels and TLB

Table 5.1: CPU table 1

ARM Cortex-A53				
DTLB		Levels	L1	L2
Entries	10	Size	32 KB	512 KB
Pages Mode	4 KB	Line Size	64 B	64 B
Associativity	32-way	Associativity	4-way	16-way
Miss Penalty	2	Latency	3	15

Table 5.2: ARM Cortex-A53 Cache Metrics

ARM Cortex-A57					
DTLB		Levels	L1	L3	UL3
Entries	32	Size	32 KB	1 MB	64 MB
Pages Mode	4 KB	Line Size	64 B	64 B	64 B
Associativity	32-way	Associativity	2-way	16-way	16-way
Miss Penalty	7	Latency	5	18	60

Table 5.3: ARM Cortex-A57 Cache Metrics

IBM Power8						
DTLB (D-ERAT)		Levels	L1	L2	L3	UL4
Entries	48	Size	64 KB	512 KB	8 MB	16 MB
Pages Mode	64 KB	Line Size	128 B	128 B	128 B	128 B
Associativity	48-way	Associativity	8-way	8-way	8-way	8-way
Miss Penalty	11	Latency	4	12	27	45

Table 5.4: IBM Power8 Cache Metrics

5.1.2 DRAM Metrics

The Memory System is portable, as well as, the CPU. The user can modify and experiment with his/her own choices. The Device Ini and System Ini of DRAMSim2 contain many variables which can be modified. A modified Micron MT40A1G4HX-083E DDR4 SDRAM[Micron site] it is used in the current thesis.

The original model has by default 4 GB of total storage, but the benchmarks that the current thesis used had an average less than 1 GB footprint. So, the Rows and Columns were modified, in order to shrink the original model to 1 GB of total storage.

Some of the metrics of modified DDR4 Micron 1G 16B x4 sg083E model are contained in Device ini file and presented below:

Name	Value
NUM BANKS	16
NUM ROWS	16384
NUM COLS	8192
DEVICE WIDTH	4
tCK	0.85 ns
REFRESH PERIOD	7800 ns
CL	16
tRAS	32
tRCD	16
tRRD	4
tRC	48
tCMD	1
Vdd	1.2

Table 5.5: DRAM Metrics - Modified DDR4 Micron 1G 16B x4 sg083E

5.1.3 Cloud Benchmarks

Cloud benchmarks are certain tests which are applied to cloud computing services offered by companies and organizations. More broadly speaking, we would say that benchmarks are not merely tests, but they are considered best practices for certain components of the Cloud computing which are examined. Benchmarking is used for performance analysis and, from every aspect and type of computing, one can discover that there are many benchmarking tests (pieces of software) available, proprietary or not, in order to be utilized by the tester. For example, let us refer to the geekbench product, which is a commercial CPU benchmarking solution. Geekbench has got 27 different workloads (tests) at its disposal: namely, 13 integer tests, 10 floating-point tests and, lastly, 4 memory tests[1]. There are many cloud benchmarks of the kind, such as those developed by large market-share holders such as Google (PerfKitBenchmarker) and Oracle, and other companies such as Upcloud. The one used in this thesis is CloudSuite[10, 25].

5.1.3.1 Cloudsuite

The CloudSuite employs certain specific criteria in order to evaluate the performance and efficiency of the cloud services to be assessed. The scale-out services (as opposed to the traditional ones) examined by the Suite are the most popular ones [10] and are the following:

a) Data serving

b) Media streaming

c) Web search

Furthermore, some additional ones, as included in CloudSuite 3.0, are:

g) Data analytics

h) Graph analytics

i) In-Memory analytics

j) Data caching

The examination of the aforementioned can depict the architectural behavior of the system, database and data transport performance, storage velocity, media files transmission, fault tolerance, web services throughput and performance scalability. All the benchmarking tests should be performed when the system has reached a steady state[10, 25].

Lastly, let us refer to two elements which have especially been utilized during the course of this thesis. These are Data caching and Media streaming.

Data Caching

Data Caching is an online benchmark; it is meant to constritute a performance criterion of how well the system does when it is presented with an arbitrary amount of online requests. The manner in which it functions emulates Twitter users and memcached software is used to this end. The performance metrics put to use are the number of requests served per second and the latency percent for the total of these requests. The idea here is that, since data access is slow, and every failure to reach to cache and retrieve a piece of data will result in a disk request, the metrics are thought to effectively measure the performance of data caching, which results in quick data access.

Media Streaming

Media Streaming benchmark, on the other hand, is still an online benchmark, but referring to multimedia access performance. An HTTP Connection is made at first and, then, a flexible total of video-streaming requests is sent to the servers. Videos of all types and quality can be accessed and the performance metrics calculated are the streaming bandwidth and the average delay for the reply to each request that was sent.

5.2 Experimental Process

In that section the experimental process strategy is going to be described. The result reliability achievement is the main problem of an experimental process design. We developed on the Simulator the functionality of the Skip mode, because of the time consuming simulation as a result of the large overhead over the benchmark execution time[31]. Skip mode, in essence, is a sampling method, consequently, the existence of the statistical error ε cannot be hidden; it has to be determined, as well as the sampling rate, to make the reader able to evaluate the result's reliability. The sampling rate, as a function of "how many samples", describes "when", a measurement, or in our case a piece of simulation, must be applied for the result reliability maintenance.

We choose to use a simulation sample size of 2 Billion benchmark instructions, our experiments The trade-off simulation sample size vs simulation time had lead us to that choice. The sample has to be large enough to let the benchmark unfold its true workload execution, but always in the context of paying the least useful overhead. So that, the 2 Billion was estimated as a good trade-off, because that size is quite over the usual benchmark execution and simulation threshold, which is 1 Billion (eg the Ramulator [18] evaluation uses 10M main memory accesses which approximately corresponds to 1% miss rate per Billion Instructions), and also the overhead was not excessively high. The overhead affection was experimentally observed.

The simulation result of the 2 Billion sampling must be scaled up to the whole benchmark size, in order to produce the final result. At this point, the final result must be accompanied with the corresponding ε.

To calculate the ε, a complete simulation, without sampling, is the one way. But as a result of the time limitations, imposed by the complete simulation, (expected duration of Media Streaming$\sim$ 90 days = 3 months or non-stop data flow on Datacache), the solution of just a 4 times denser simulation has been chosen. A simulation with higher density will not provide us with an accurate ε, but with just an indication of our sample selection accuracy level.

Sample Selection: The samples must be uniformly distributed along the whole benchmark instructions, to make the sampling representative. A simple Workload Profiling Pintool was developed to instrument the benchmark and produce a last level cache miss profile representation. The functionality of that Pintool is quite simple. The whole benchmark is simply instrumented, in order to profile the last level cache miss number of each window. That number is recorded to a .csv file, so that profile can be depicted in a chart.

The offered knowledge of a last level cache miss profile chart give us the ability to design the simulation process, in order to distribute in a uniform and statistically correct way our simulation samples.

In order to achieve the corresponding and appropriate sampling distribution, each benchmark has to be utilized separately.

Finally we have to say, that we have failed to take the Simulation results of the Media Streaming Benchmark because after the final video streaming the Client terminates and the Server restarts,making the problem of deleting the log files, as a new Server container starts, after deleting the old one automatically.

Only the DataCaching Profiling and Utilization with 4GB memory size is presented bellow, as all the other profiles are the same.

5.2.1 Data Caching Profiling and Utilization

As we can see in figure 5.1, the diagram cannot be divided in any areas. We observe that the majority of the values is 0,4 MPKI, except of the initialization phase ones (the first 4 windows)

Due to that stability it was prefered to simulate a solid area. In that benchmark we use 200 Million Instructions sized Windows, subsequently, to perform a 2 Billion Instructions Simulation, we need 10 Windows (samples).

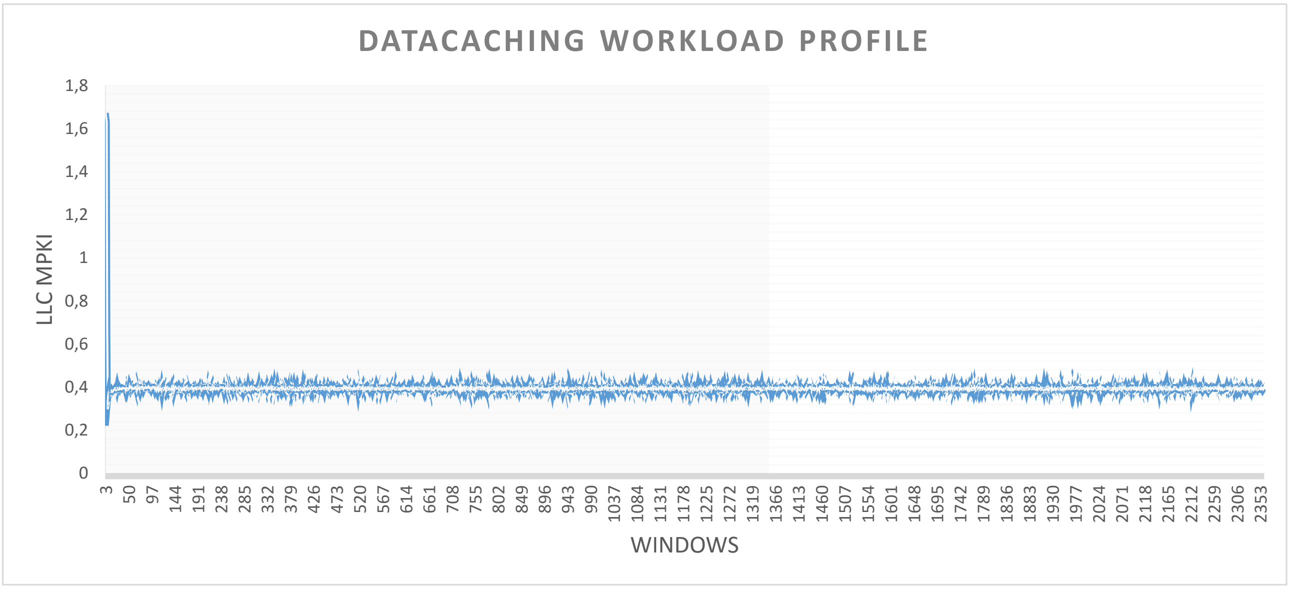


Figure 5.1: DataCaching Power8 workload analysis.

5.2.2 Simulation Scenarios

The final issue of Experimental Process Design is choosing the Simulation Scenarios. These scenarios describe the Memory Disaggregation Percentage, or, in other words, how the whole Simulated Disaggregated System makes use of its memory. For example, if the system has to execute a 16 GB memory footprint application, which amount is going to be delivered in Local, and which one in the Remote Memory.

The Memory Disaggregation Percentage is a dynamic process result, through which, the system will be able to choose between several factors, such as the latency, the availability etc. In our approach for simplicity reasons it was assumed that the Local Memory usage percentage, as well as, the Remote are static and predefined. That assumption had been lead to the different Scenarios we used:

1. Local: 100%, Remote: 0%
2. Local: 75%, Remote: 25%
3. Local: 50%, Remote: 50%
4. Local: 25%, Remote: 75%

To implement these scenarios many different Disaggregated Latencies have been used. The latency range is 500-2000 ns with a step of 250 ns. The Disaggregated Latency had to be translated in Memory Cycles. The used Memory Model has a 0.85 ns Cycle, so we have the latencies in cycles below:

Latency in ns	Latency in Cycles
500	588
750	882
1000	1176
1250	1470
1500	1764
1750	2058
2000	2352

Table 5.6: Nanosecond to Memory Cycles Disaggregated Latency Correspondence

5.3 Experimental Results

The experimental results are presented per Benchmark, CPU and Simulation Scenario. Firstly are illustrated the results of 2GB Data Caching benchmark with Power 8, then with Cortex-A53 and finally with Cortex-A53. Subsequently, the results of 4GB, 8GB, 16GB, and 32GB of Data Caching Benchmarks are presented in the same concept.

A latency oriented chart (left column), as well as a bandwidth one (right column), corresponds for each Scenario. The latency oriented chart shows the, positive or negative, Overhead in total execution time over the 100-0 Scenario in a stack bar per Disaggregation Latency Step. As total execution time, is defined the addition between the Local and Disaggregated execution time.

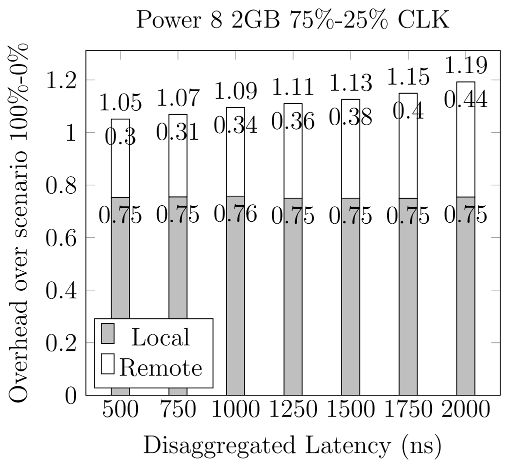
With that kind of chart, we are able to observe how the Disaggregated nature and behavior differs the benchmark execution time compared with the non Disaggregated (scenario 100-0). The point that can be observed is that, the charts depict the expected increase on the execution time. As the Disaggregated Latency increases, so does the Overhead. That fact is the expected Disaggregated effect over the total execution time.

The second kind of chart is about the Bandwidth. In that line chart the Average Memory Bandwidth results in absolute values are depicted, also per Disaggregation Latency Step. Here we can observe that as the Disaggregated Latency increases, the Bandwidth falls. That phenomenon also validates the right Disaggregated behavior of the DiMEM Simulator because the Disaggregated Latency enlarges the time distance between the Memory Accesses and as a result we can observe the Bandwidth loss.

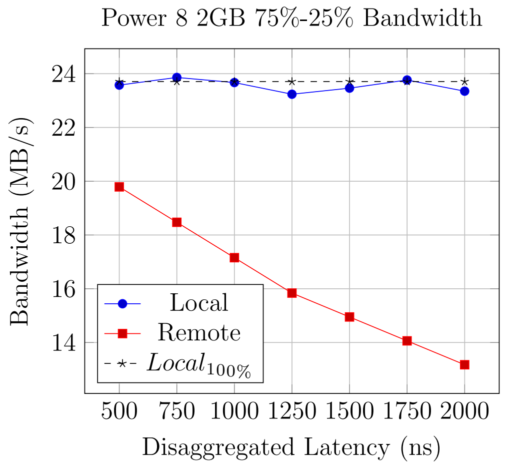
5.3.1 Dense Data Caching Experimental Results

As noticed earlier, a denser experiment for Data Caching has been performed, in order to evaluate that Benchmark's experimental process. The Data Caching Benchmark profile shows a stability on 0,4 MPKI.

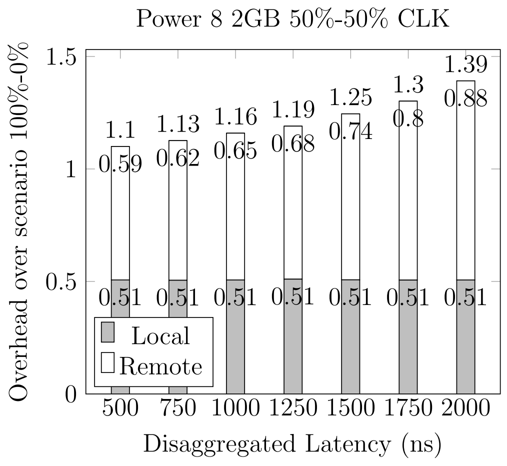
The Dense Experiment results show 0.078% divergence in comparison with the 2 Billion Instructions Experiment. The above results were expected due to the specific behavior of the workload profile.



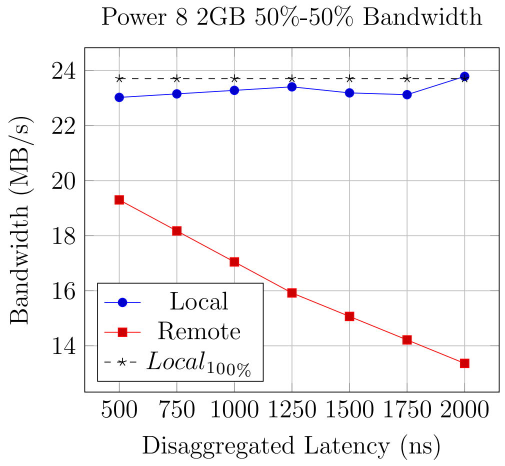
(a)



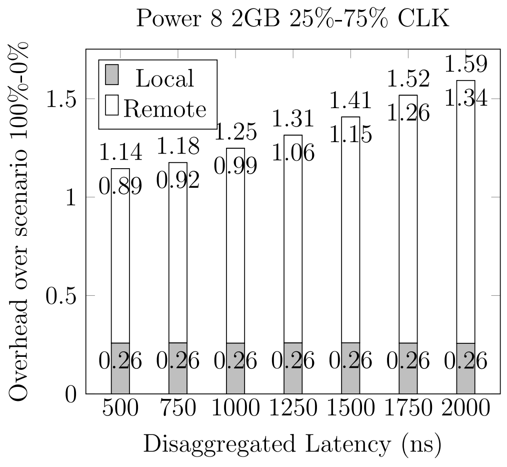
(b)



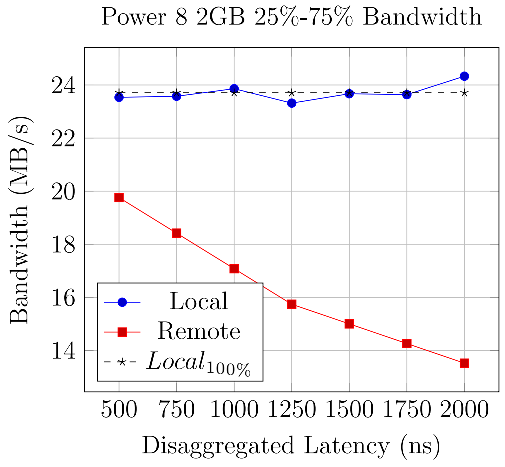
(c)



(d)

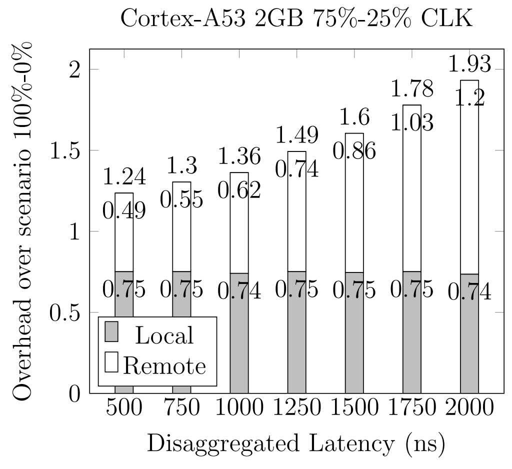


(e)

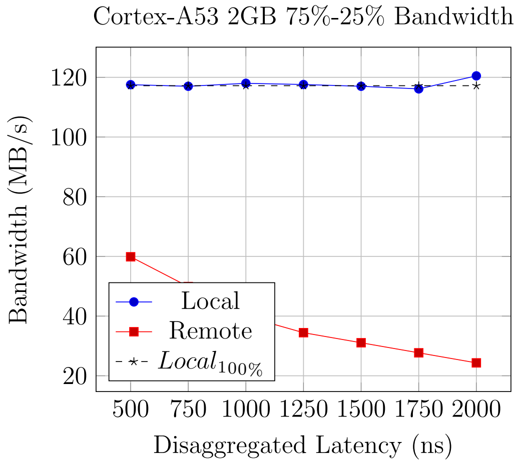


(f)

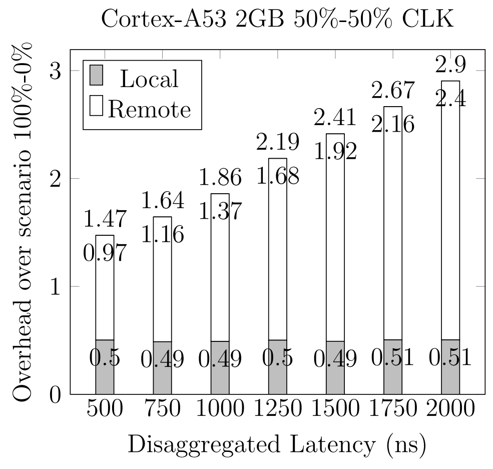
Figure 5.2: Power 8 DataCaching 2GB Results



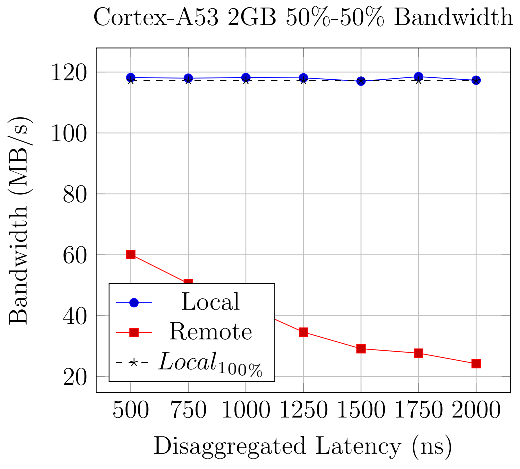
(a)



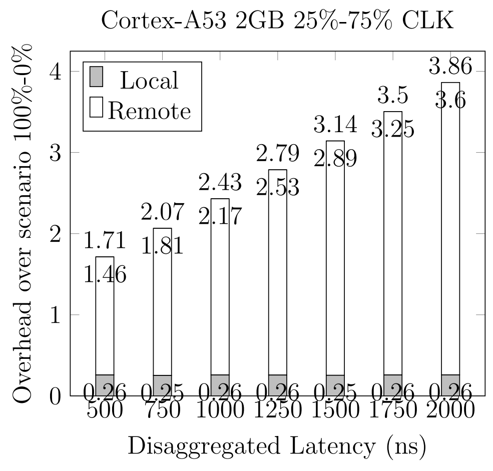
(b)



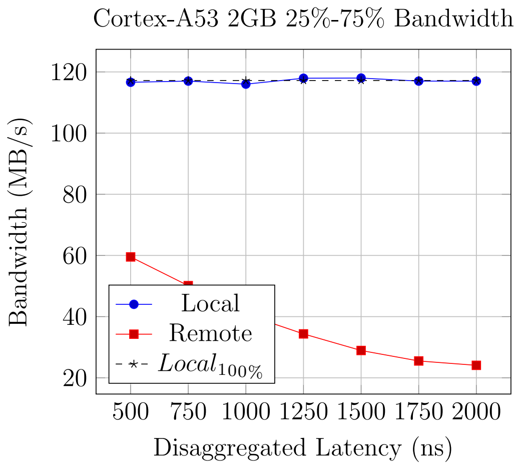
(c)



(d)

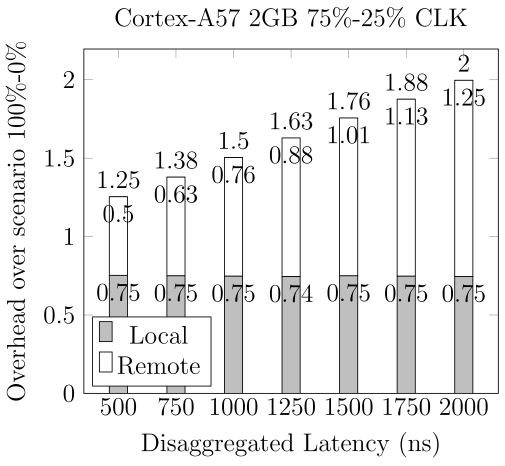


(e)

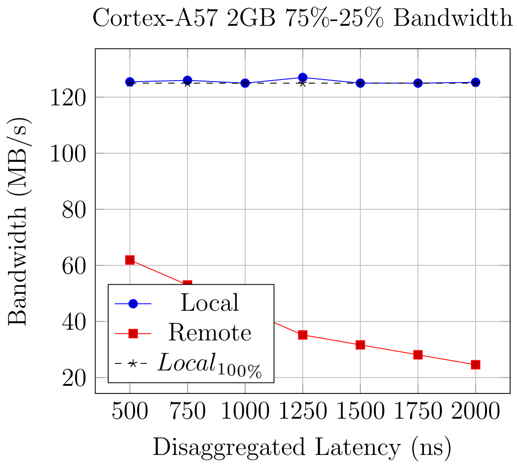


(f)

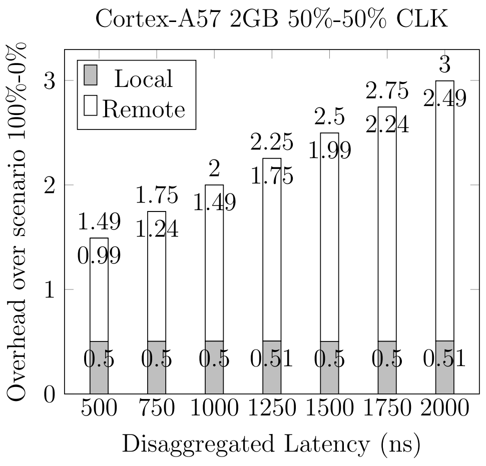
Figure 5.3: Cortex-A53 DataCaching 2GB Results



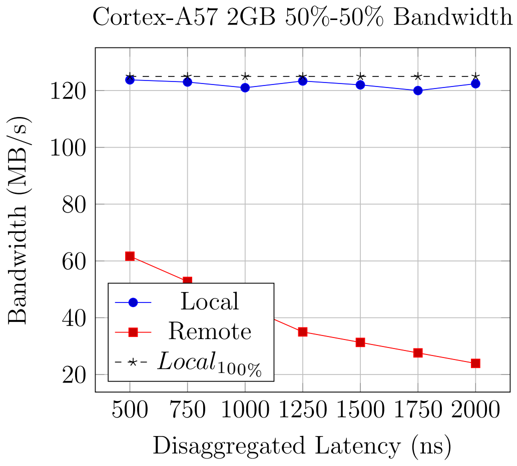
(a)



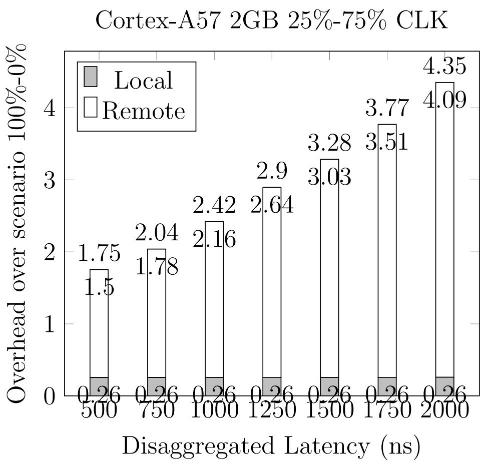
(b)



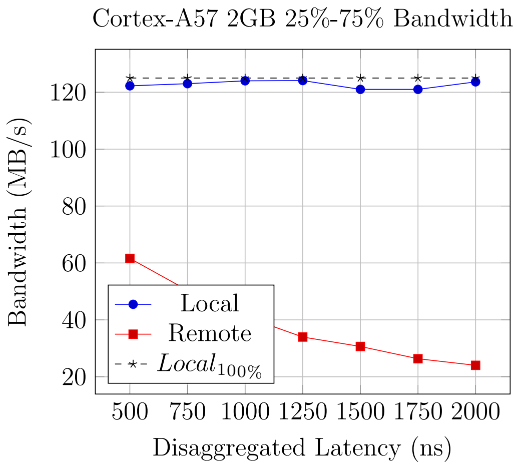
(c)



(d)

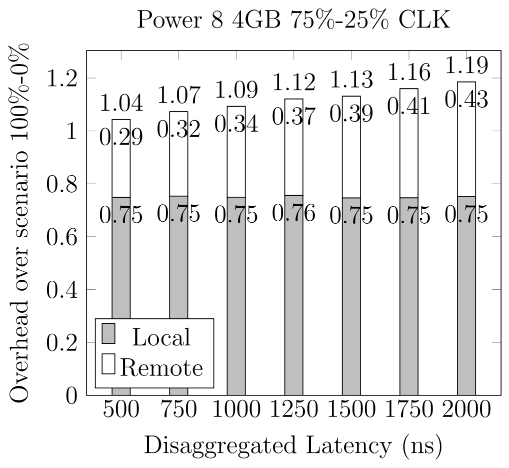


(e)

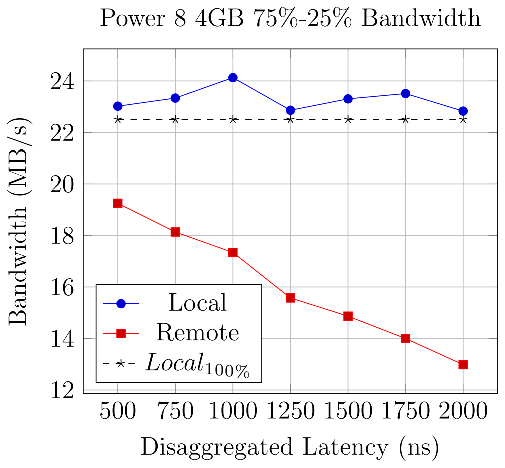


(f)

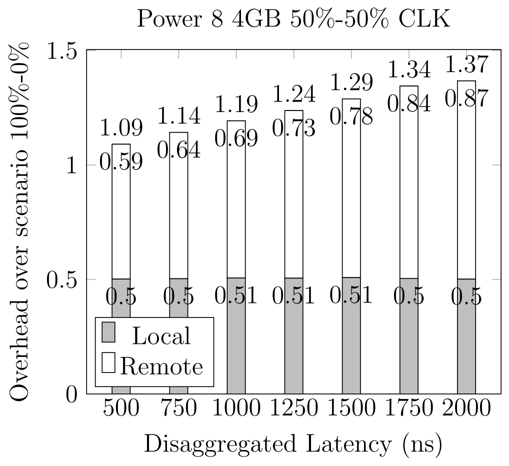
Figure 5.4: Cortex-A57 DataCaching 2GB Results



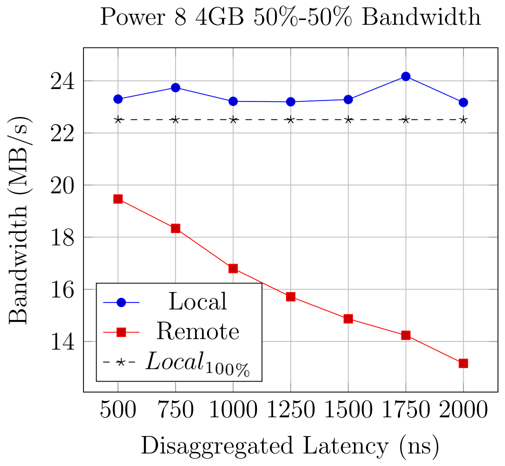
(a)



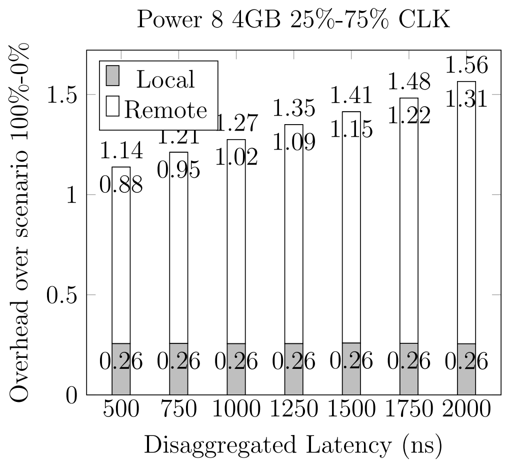
(b)



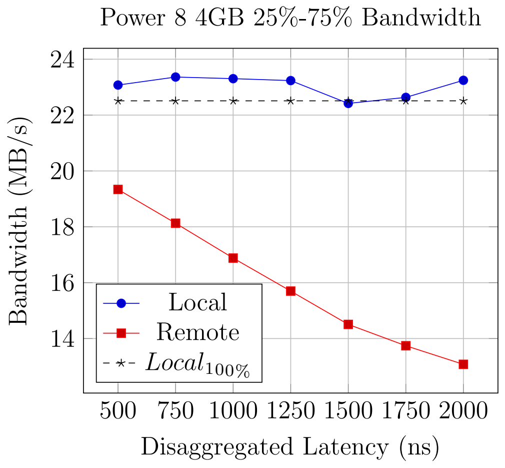
(c)



(d)

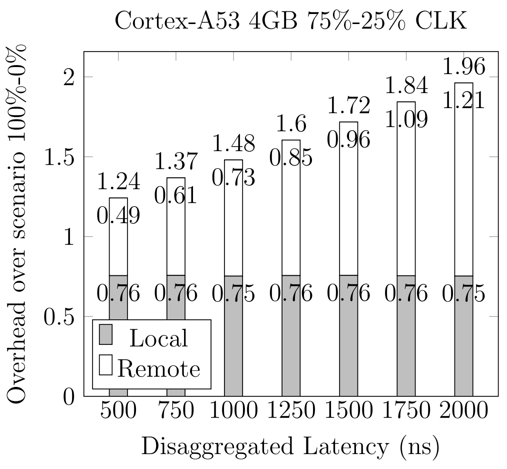


(e)

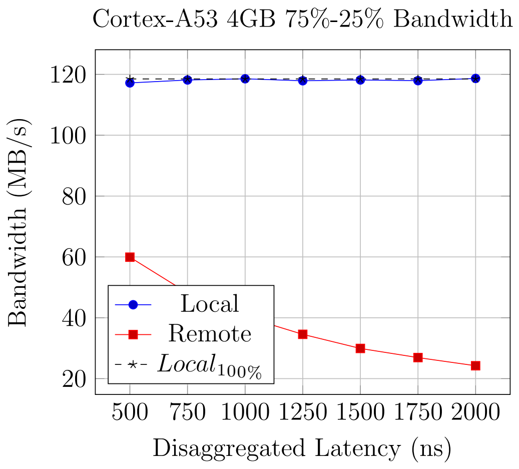


(f)

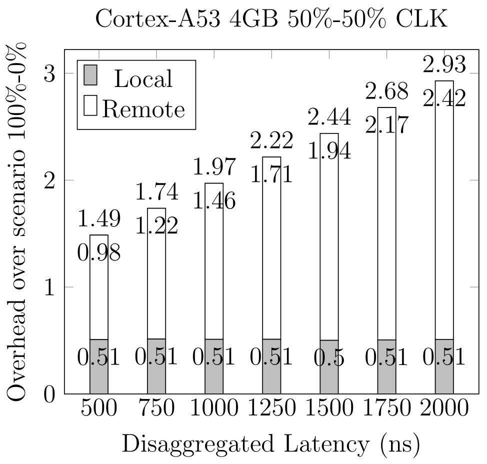
Figure 5.5: Power 8 DataCaching 4GB Results



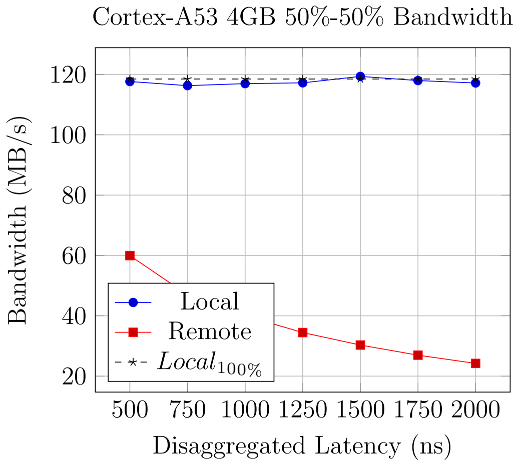
(a)



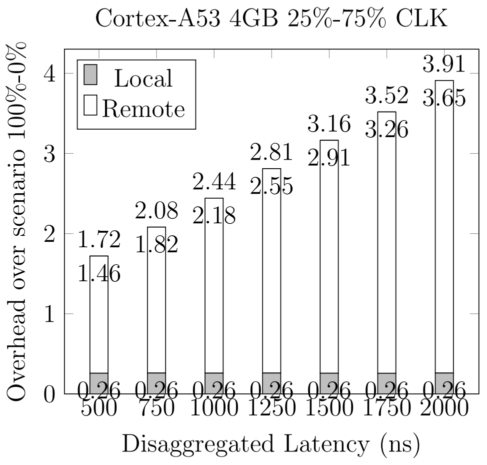
(b)



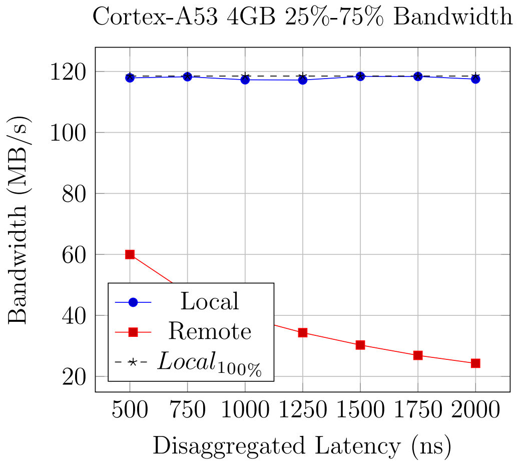
(c)



(d)

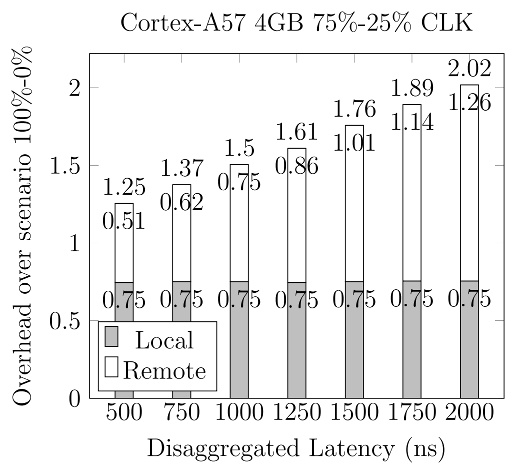


(e)

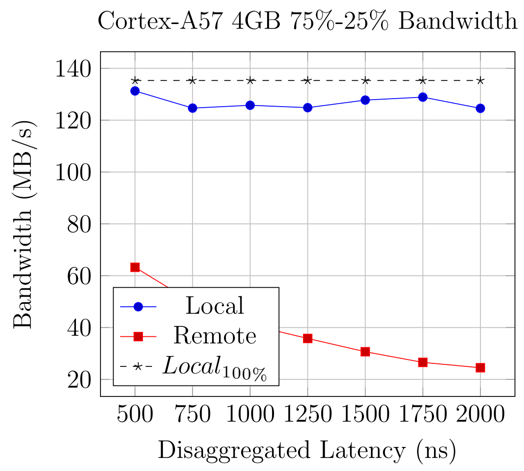


(f)

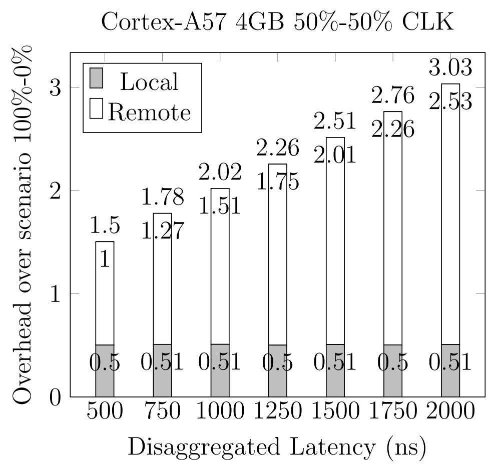
Figure 5.6: Cortex-A53 DataCaching 4GB Results



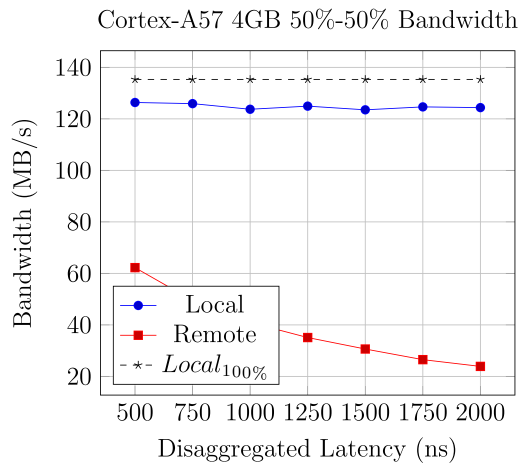
(a)



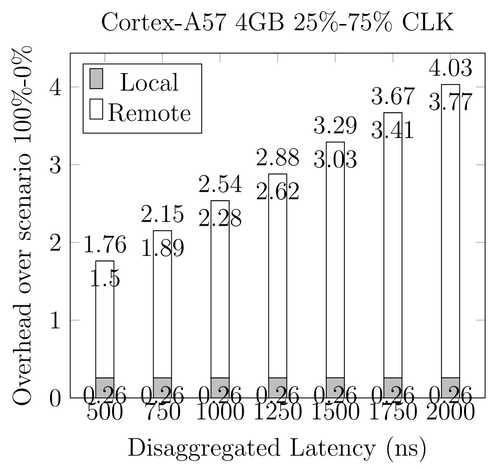
(b)



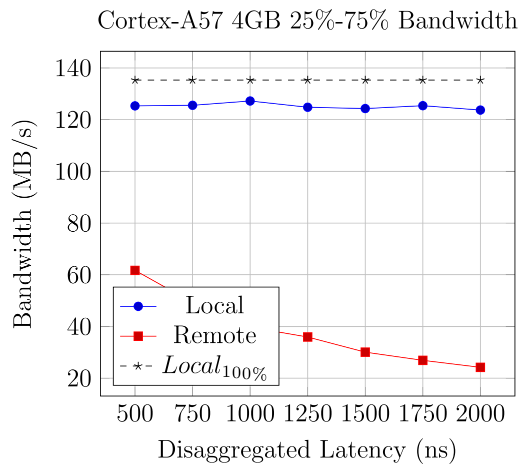
(c)



(d)

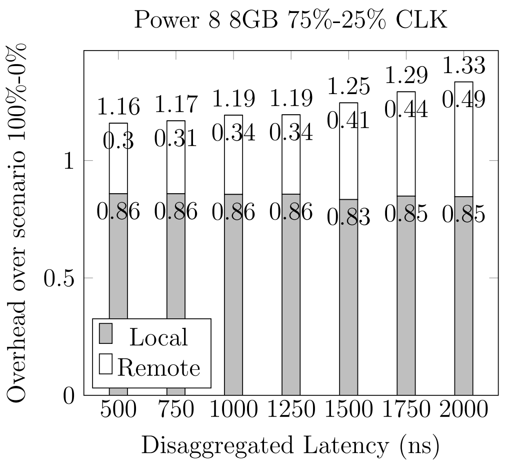


(e)

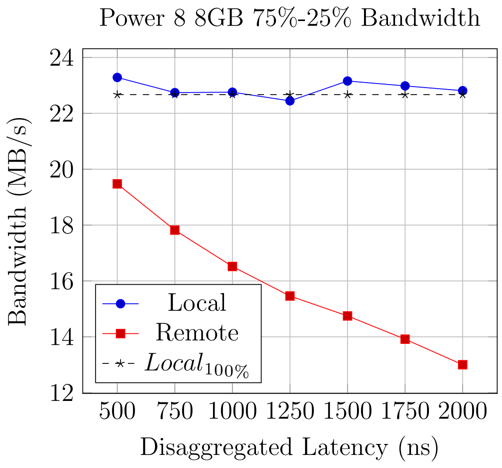


(f)

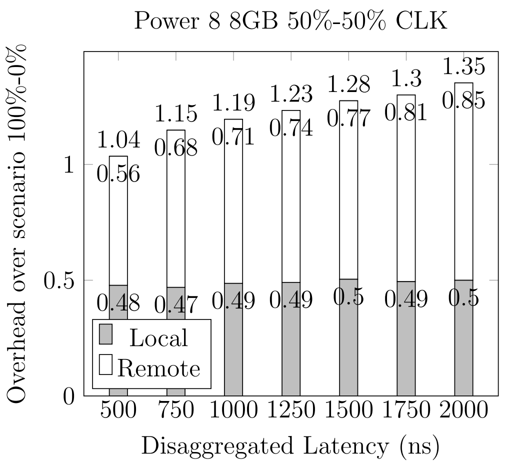
Figure 5.7: Cortex-A57 DataCaching 4GB Results



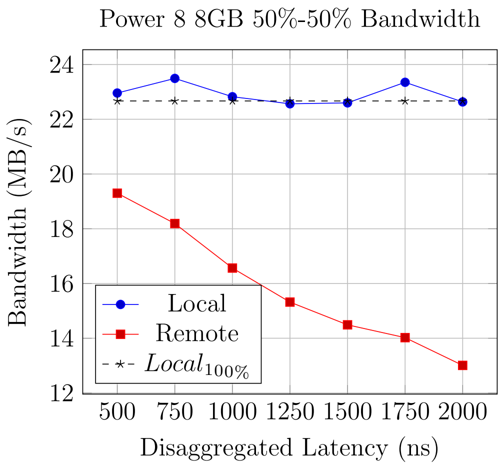
(a)



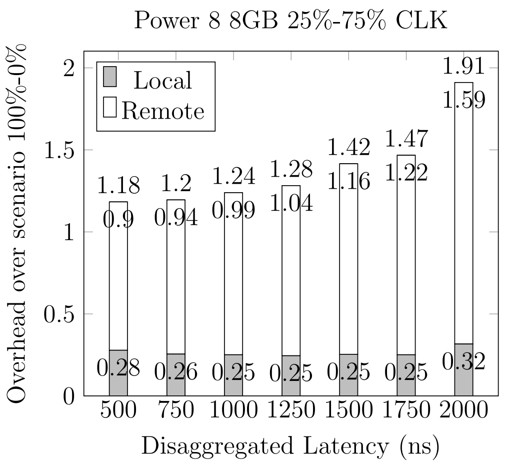
(b)



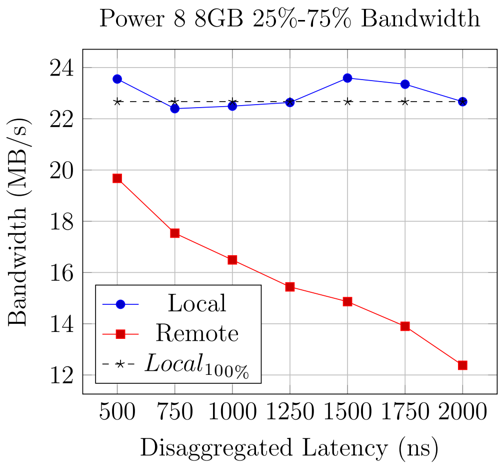
(c)



(d)

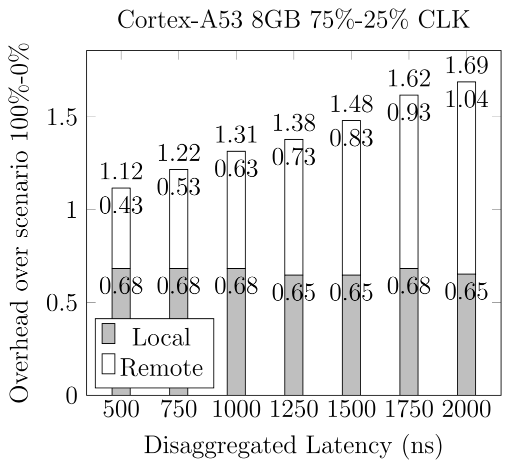


(e)

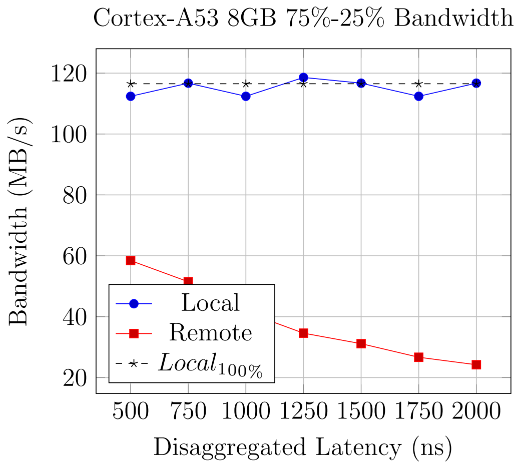


(f)

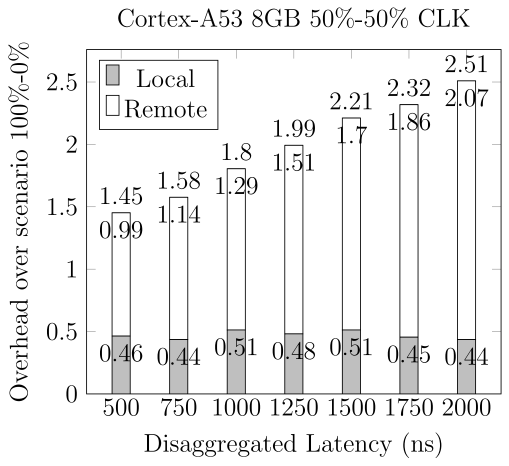
Figure 5.8: Power 8 DataCaching 8GB Results



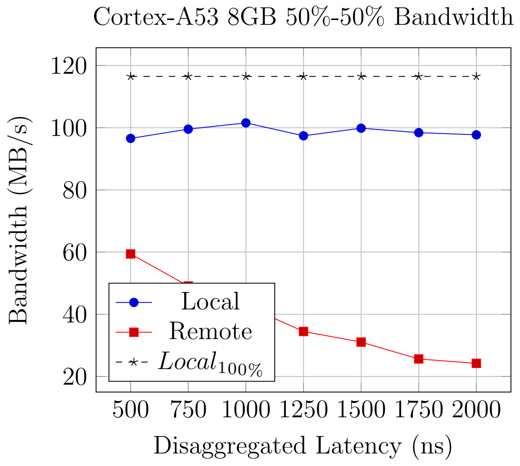
(a)



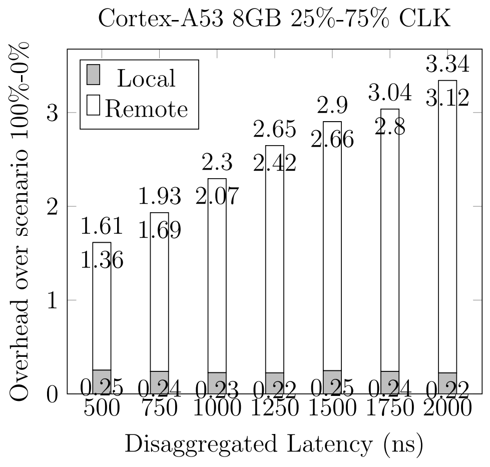
(b)



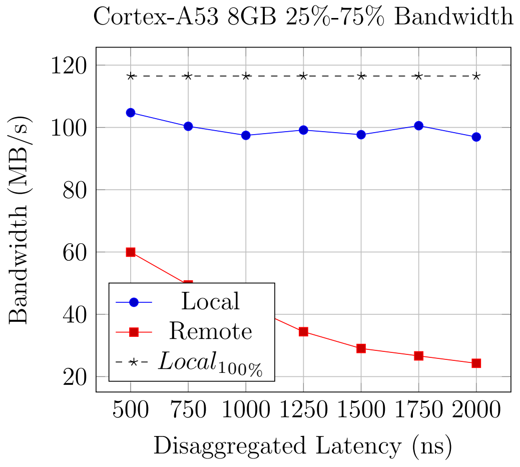
(c)



(d)

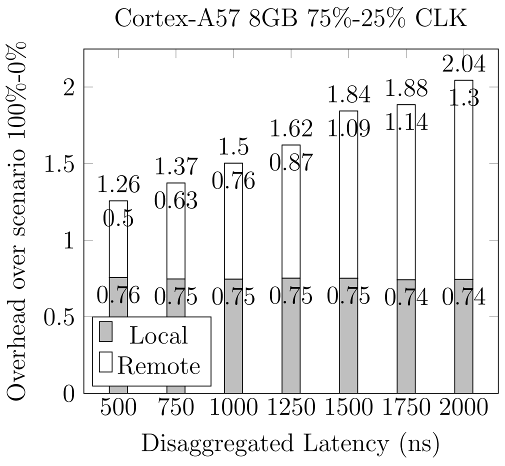


(e)

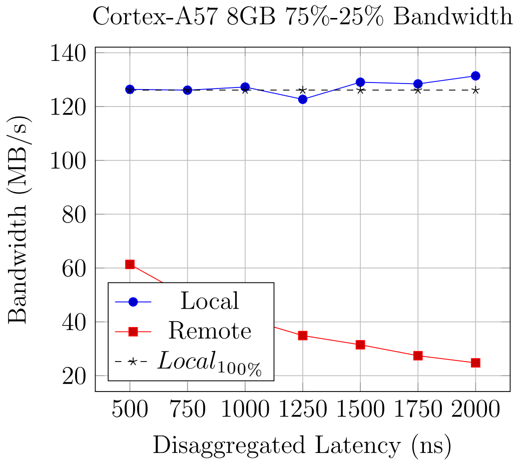


(f)

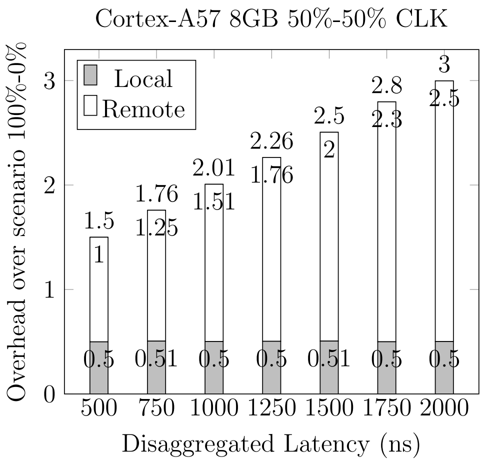
Figure 5.9: Cortex-A53 DataCaching 8GB Results



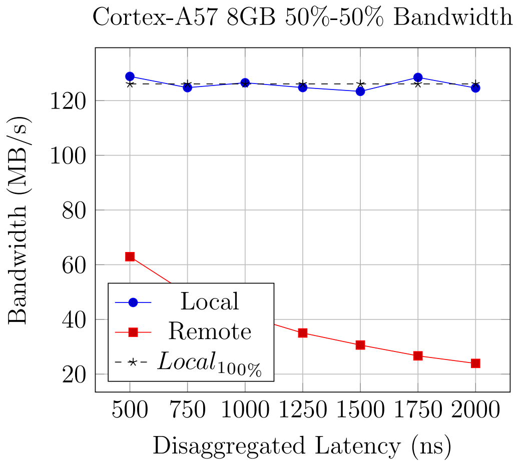
(a)



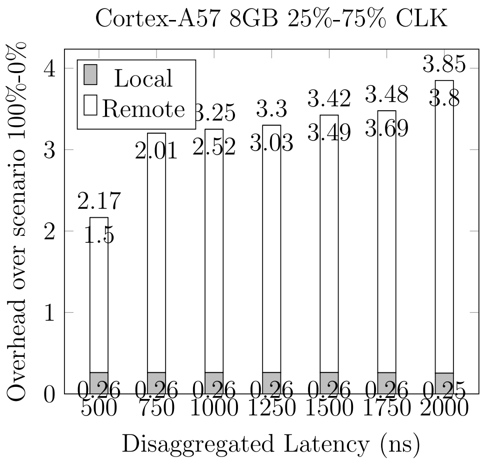
(b)



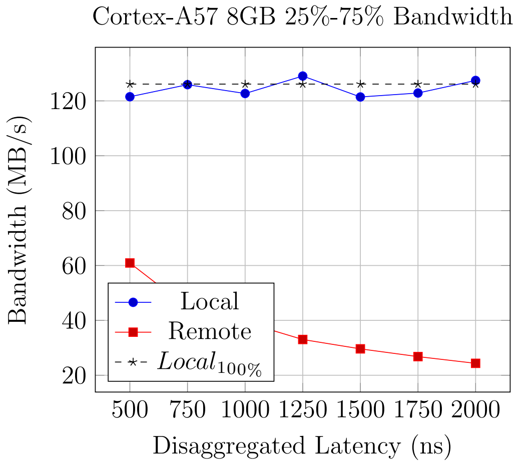
(c)



(d)

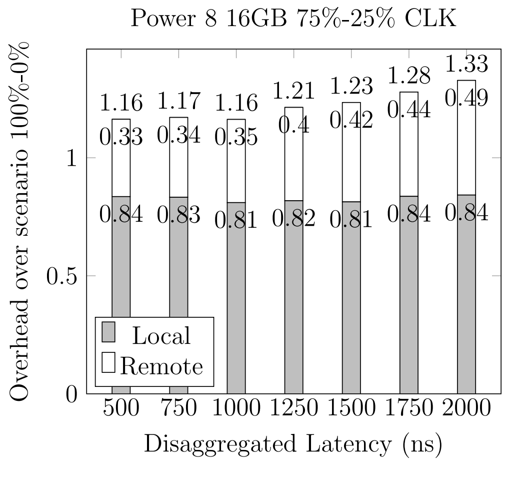


(e)

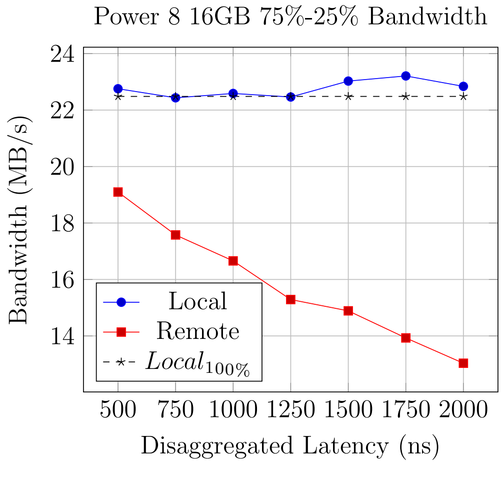


(f)

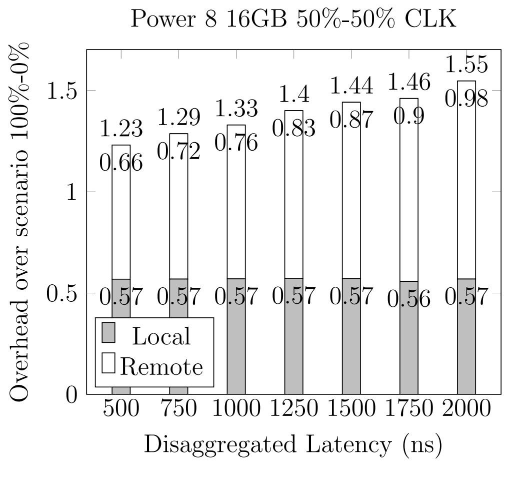
Figure 5.10: Cortex-A57 DataCaching 8GB Results



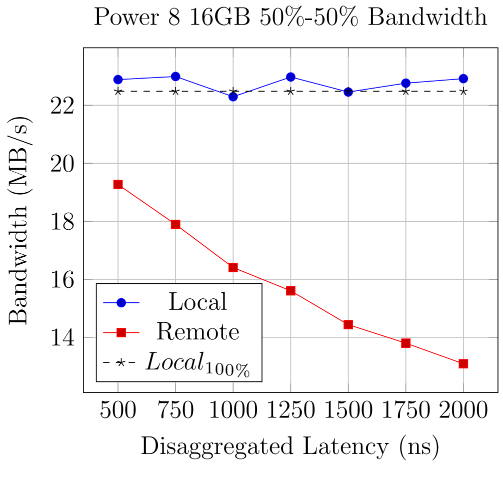
(a)



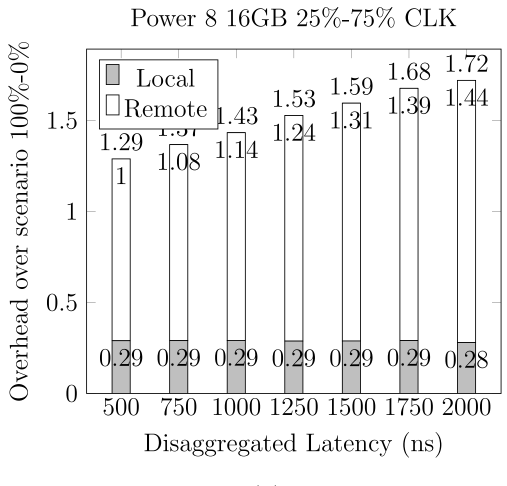
(b)



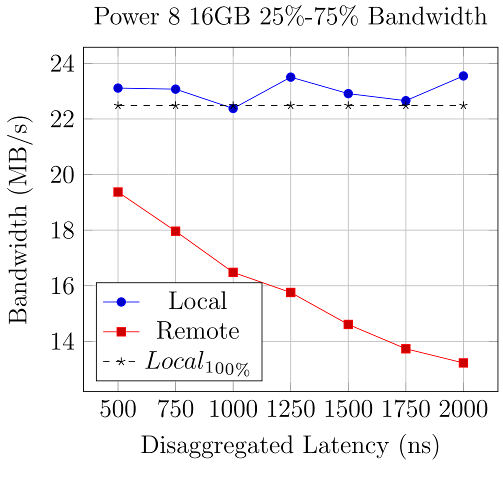
(c)



(d)



(e)



(f)

Figure 5.11: Power 8 DataCaching 16GB Results

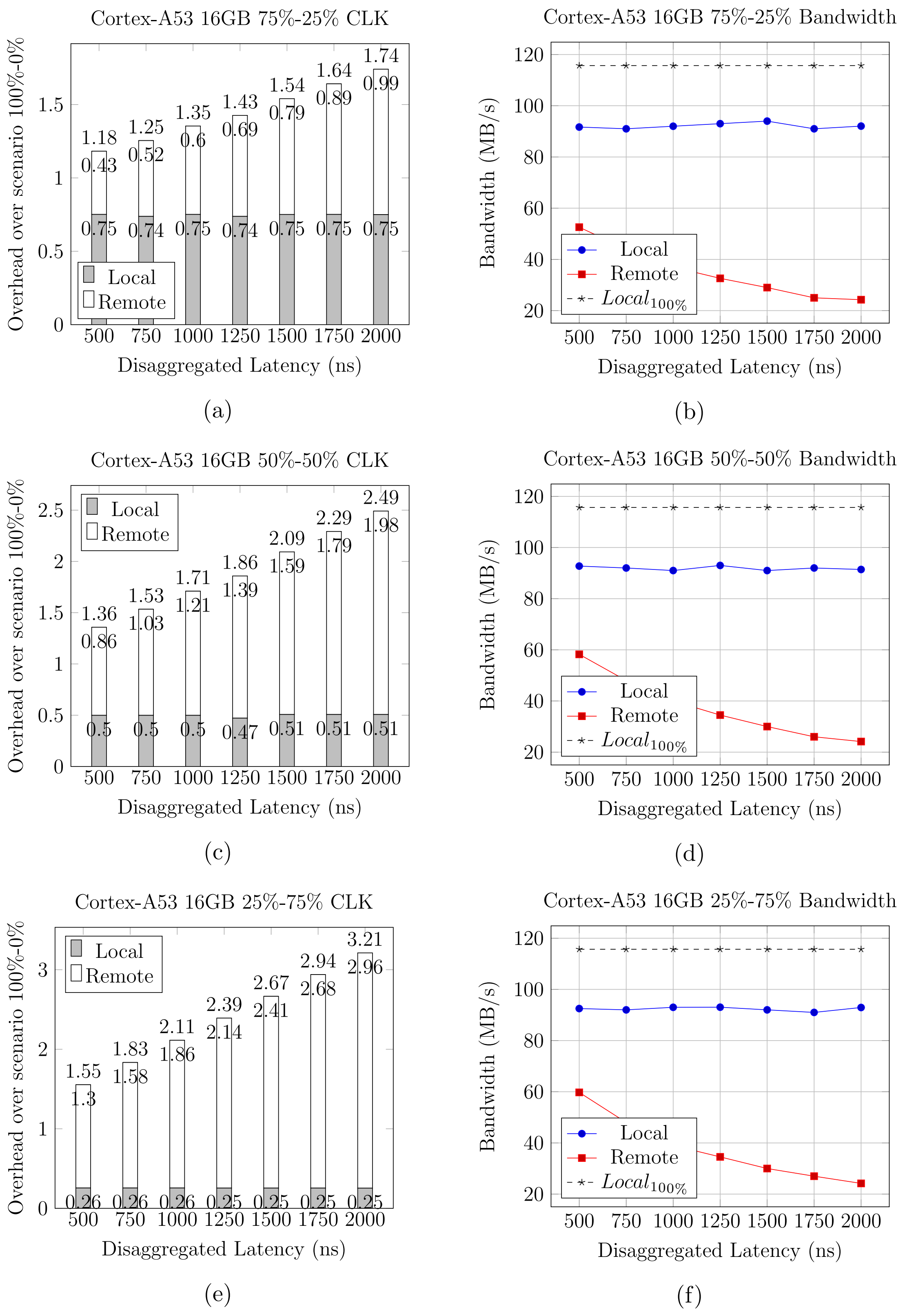
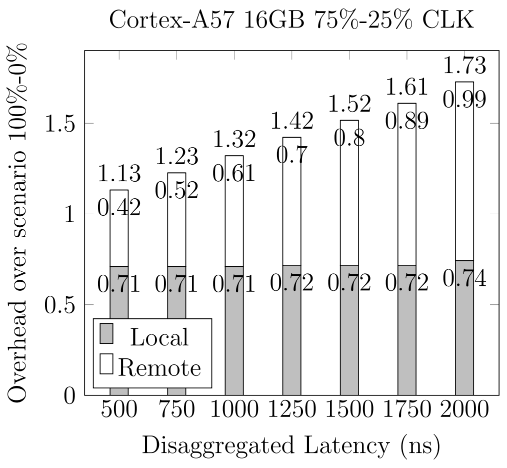
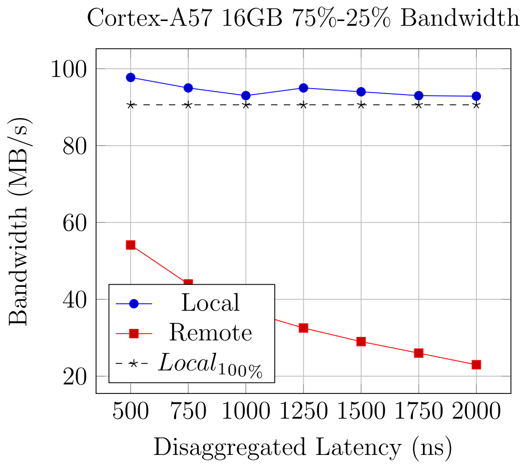


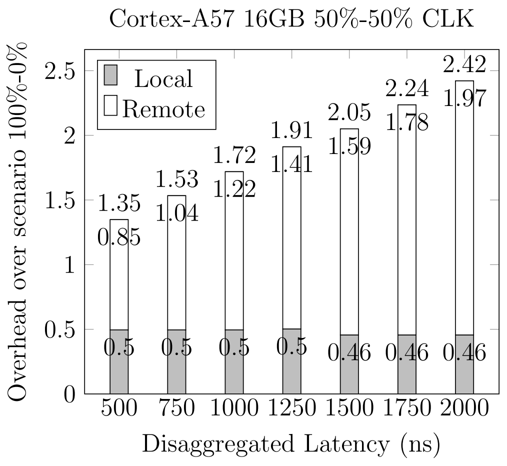
Figure 5.12: Cortex-A53 DataCaching 16GB Results



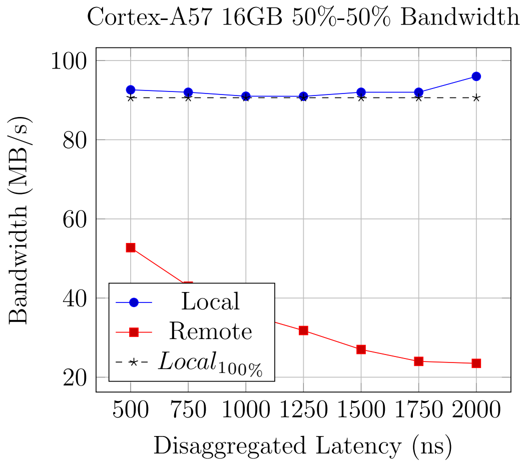
(a)



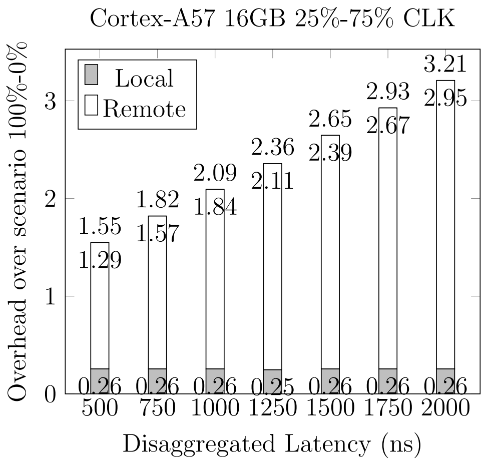
(b)



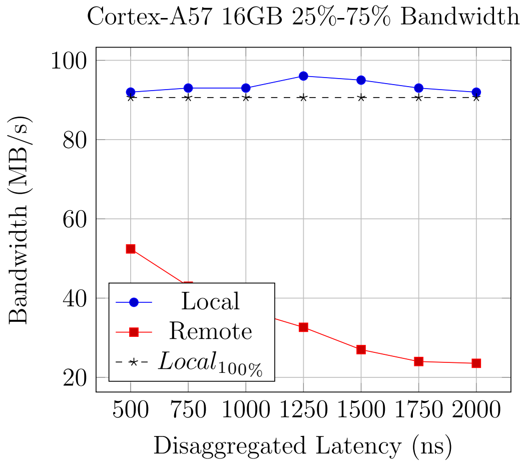
(c)



(d)

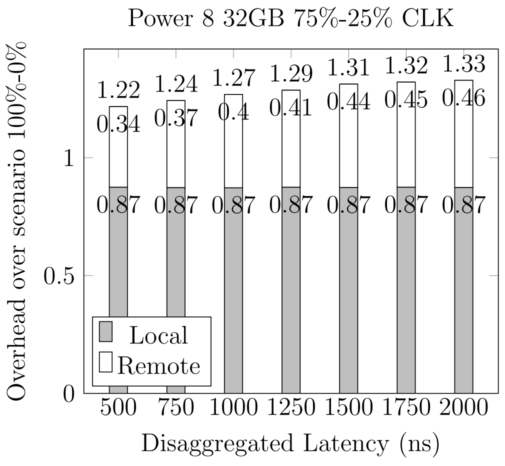


(e)

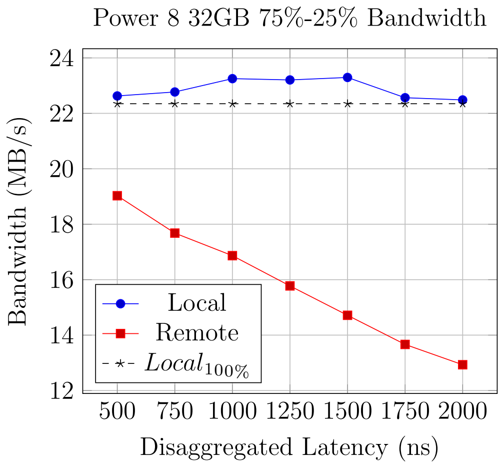


(f)

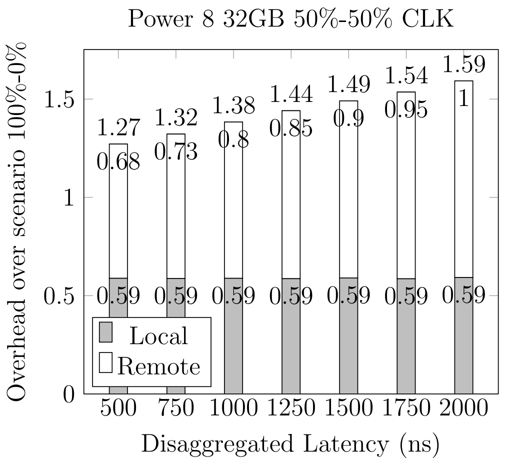
Figure 5.13: Cortex-A57 DataCaching 16GB Results



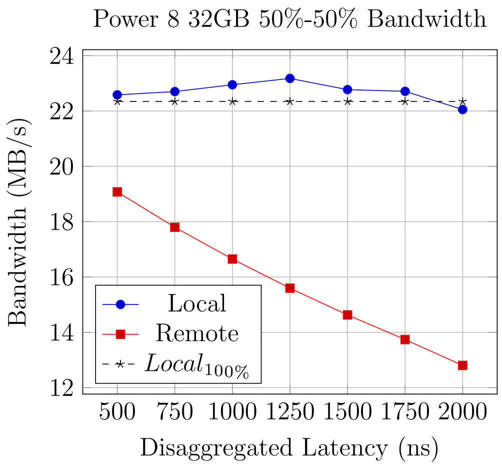
(a)



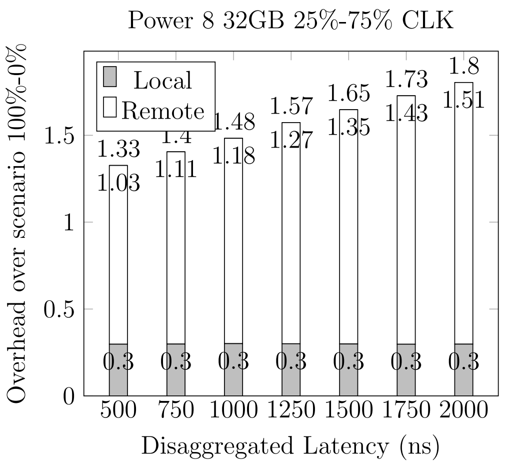
(b)



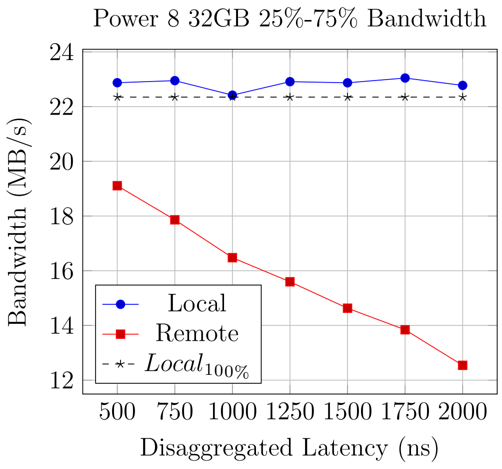
(c)



(d)

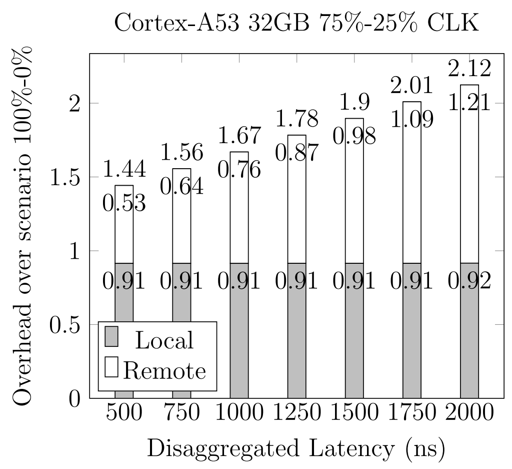


(e)

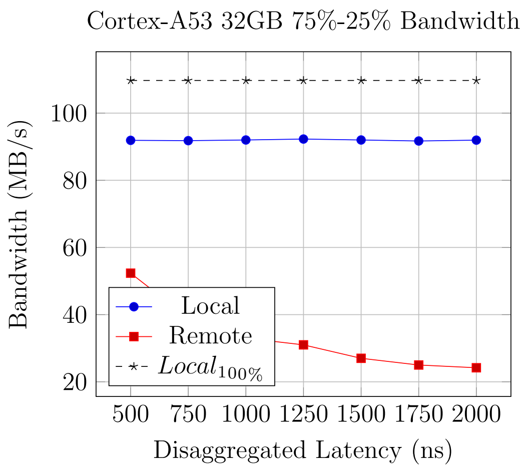


(f)

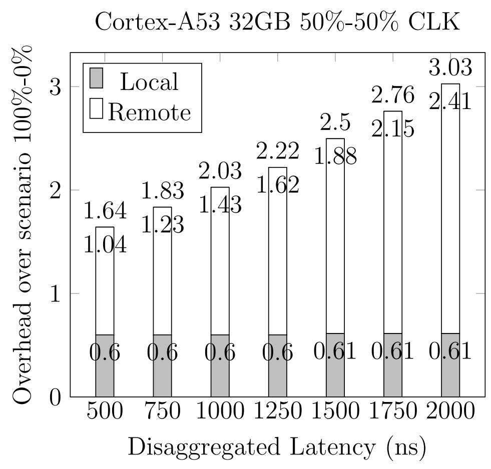
Figure 5.14: Power 8 DataCaching 32GB Results



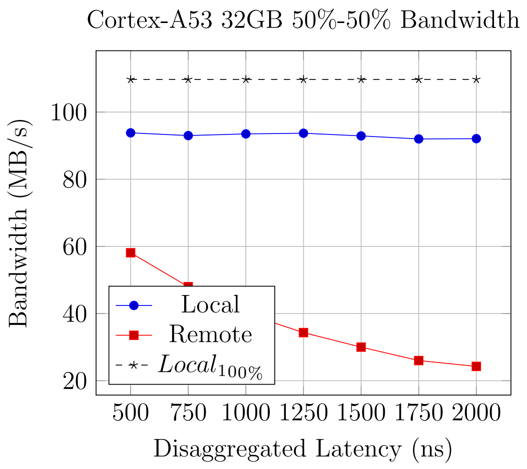
(a)



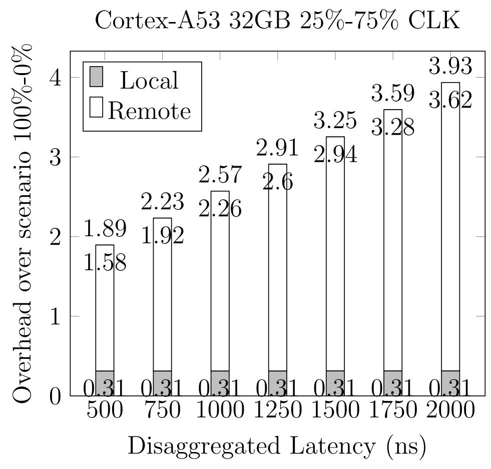
(b)



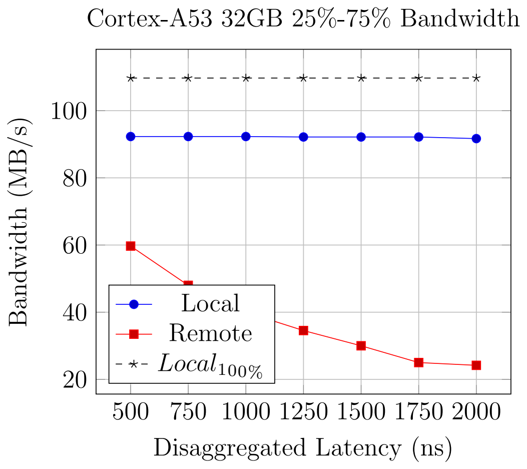
(c)



(d)

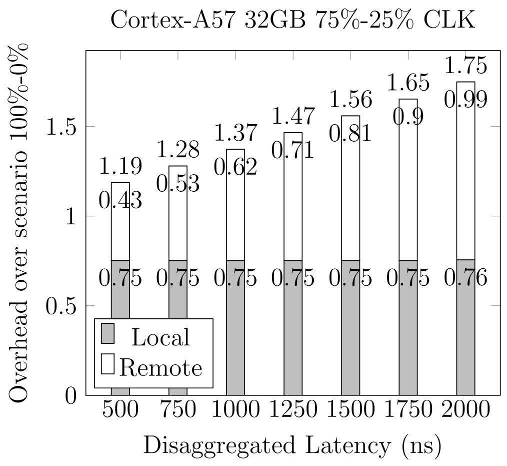


(e)

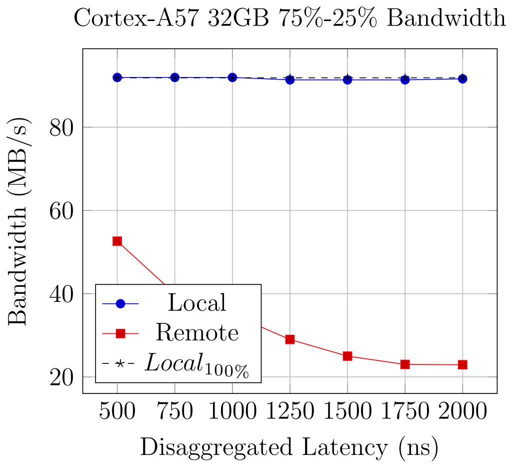


(f)

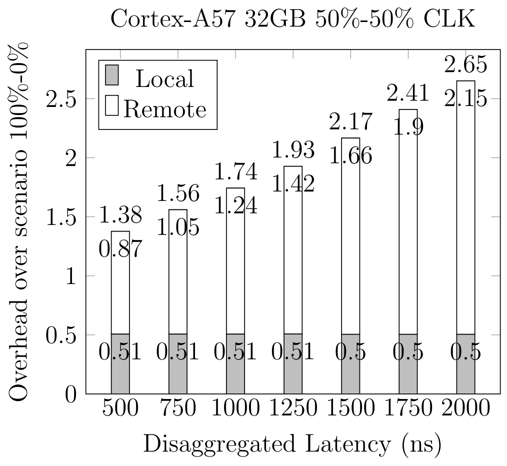
Figure 5.15: Cortex-A53 DataCaching 32GB Results



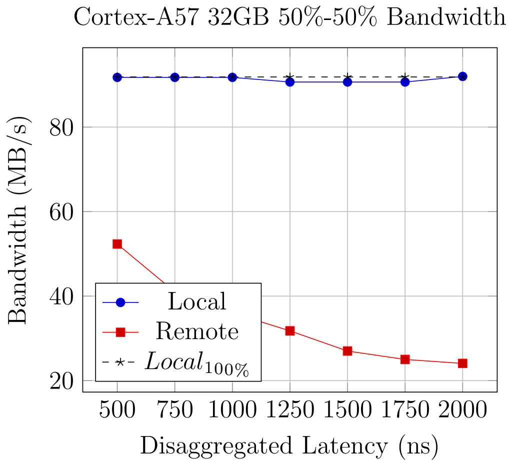
(a)



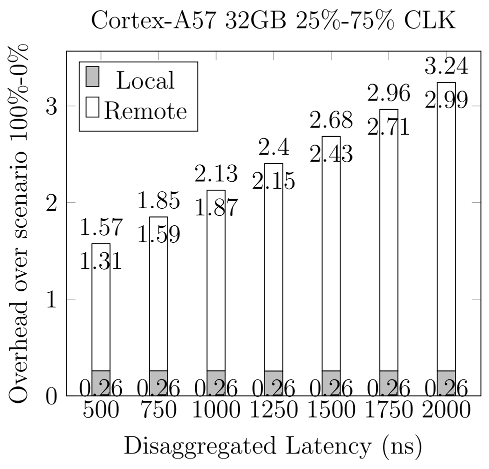
(b)



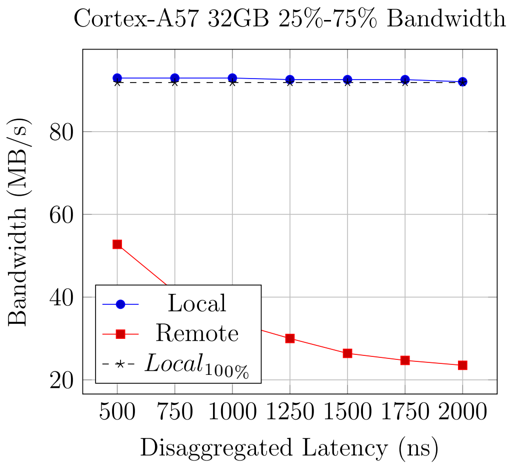
(c)



(d)



(e)



(f)

Figure 5.16: Cortex-A57 DataCaching 32GB Results

CHAPTER 6

Conclusion and Future Work

6.1 Conclusion

In conclusion we can support that we have developed a simple Disaggregated Memory System Simulator in combination with Orion Papadakis thesis[26]. Our both approach comes with a simplistic level of detail, so that it can depict the Disaggregated Memory System behavior, in an approximate way. It, also, bears the Intel Pin framework and DRAMSim2 coupling. Despite the fact that the coupling result comes along with a large overhead, it is useful for future work and study on Memory System Simulators. It can play a supporting role as a base in Disaggregated Memory System Simulators, due to the fact that a Simulator like the DiMEM Simulator has never developed before, not even in a simplistic way.

Regarding to the experimental methods, we can say that only one approach was used, due to the nature of the Benchmark. To simulate in a sampling way over a benchmark application, the profile knowledge is very important. Sampling with Nyquist theorem may be more safe, accurate and reliable, than the more custom way. However the custom way sampling is not meaningless. In the current thesis approach we observed that the chosen sampling method was accurate.

Furthermore, we observe that the Disaggregated Memory System behavior is quite depicted, either in the Latency, or in the Bandwidth results. The higher Disaggregated latency we give to the DiMEM Simulator, the clock grows, and the bandwidth falls.

Comparing a system with Disaggregated Memory with a Local-only one in Instruction and Memory Access level, the Disaggregated Memory shows worse results than the Local, because of the Disaggregated Latency overhead. Despite that fact, in our evaluation approach, we can observe the existence of a point, where the results are not so bad. At the point of Disaggregated Latency = 1000 ns, where the Step is 2x increased, we can see that the Bandwidth does not present the linear 0.5x attended fall, but it is better than that. Consequently, we can assume that while the Disaggregated Latency is bellow 1000ns, we can use Disaggregated Memory without paying for the Disaggregated Latency in the highest price, which is the 0.5x of the performance.

One more important observation is that despite the size of a the simulated memory, it is very important the simulated application to be limited in the same size from the system, otherwise we will collect wrong results.

Finally, it must be noticed, that only a *high-level* evaluation approach (for example in application level) can provide us with a clear answer about when a Disaggregated System presents a better performance than the traditional organization. In future work, we propose some ideas for further and more detailed Disaggregated Systems evaluation.

6.2 Future Work

The combined work of this thesis with the Orion Papadakis' ones[26] leads us to the following common conclusions:

1. Study of the Disaggregated Memory System benefits in an *application level.* A transparent to application memory interface is needed. That study can be done using criteria like Quality of Service (QoS) in an application level comparison between Local-only Memory System and Local-Disaggregated Memory System.
2. Main Memory Access *Dynamic Split* to Local and Remote Memory. The dynamic memory access split can be achieved by implementing an algorithm which passes a memory access to local or remote using a Memory Mapping scheme.
3. Further study and experimentation with more DRAMSim2 parameters. Include the ITLB namespace on the Simulator, as well as the I(nstruction) Cache Levels
4. Experimentation with another DRAM Simulators. Ramulator[18] may be a suitable one due to the reason that it is self-presented as the Memory Simulator with the lower run time (2.5x faster than the next fastest simulator and 2.7x faster than DRAMSim2).
5. Experimentation with latest Memory Technologies. 3D-stacked DRAM studies are aiming to overcome the traditional DRAM limits. The lights are on the Hybrid Memory Cube (HMC) which is a type of 3D-stacked DRAM because of its usability for server systems and processing-in-memory (PIM) ar-

chitecture. HMC-Sim2.0[19] and CasHMC[14] Simulators have been designed for this technology with CasHMC being a step forward due to its cycle accurate HMC modeling. Regarding to the latest Memory technologies experimentation, the DiMEM Simulator can be easily upgraded to them, through a simple DRAMSim2 library replacement, for example with CasHMC library/interface. There is no need of changing the simulation feeding process, except from the update() and addTransaction() library functions which must be replaced by the corresponding new library ones.

6. Further DiMEM Simulator development/expansion by implementing new modules alongside the already existed. These modules may be pools containing different kind of resources such as FPGAs, GPUs, Accelerators, as well as, more advanced interconnects between them.

7. More detailed evaluation of the used evaluation methods. Since, Raytrace benchmark had been sampled in specific areas, the question of which is the result reliability by using random sampling, in the same benchmark, was risen.

8. Find a way to overcome docker container constraints to simulate more Cloudsuite benchmarks.

APPENDICES

A APPENDIX

Some useful codes

pin_cache.h

```
#ifndef PIN_CACHE_H
#define PIN_CACHE_H

#include <string>

#include "pin_util.H"

/*!
*  @brief Checks if n is a power of 2.
*  @returns true if n is power of 2
*/
static inline bool IsPower2(UINT32 n)
{
return ((n & (n - 1)) == 0);
}

/*!
*  @brief Computes floor(log2(n))
*  Works by finding position of MSB set.
*  @returns -1 if n == 0.
*/
static inline INT32 FloorLog2(UINT32 n)
{
INT32 p = 0;

if (n == 0) return -1;

if (n & 0xffff0000) { p += 16; n >>= 16; }
if (n & 0x0000ff00) { p +=  8; n >>=  8; }
if (n & 0x000000f0) { p +=  4; n >>=  4; }
if (n & 0x0000000c) { p +=  2; n >>=  2; }
if (n & 0x00000002) { p +=  1; }

return p;
}

/*!
*  @brief Computes floor(log2(n))
*  Works by finding position of MSB set.
*  @returns -1 if n == 0.
*/
static inline INT32 CeilLog2(UINT32 n)
{
return FloorLog2(n - 1) + 1;
}

/*!
*  @brief Cache tag - self clearing on creation
*/
class CACHE_TAG
{
private:
ADDRINT _tag;

public:
CACHE_TAG(ADDRINT tag = 0) { _tag = tag; }
bool operator==(const CACHE_TAG &right) const { return _tag == right.
    _tag; }
operator ADDRINT() const { return _tag; }
};


/*!
* Everything related to cache sets
*/
namespace CACHE_SET
{

/*!
*  @brief Cache set direct mapped
*/
class DIRECT_MAPPED
{
private:
CACHE_TAG _tag;

public:
DIRECT_MAPPED(UINT32 associativity = 1) { ASSERTX(associativity == 1)
    ; }
```

```

VOID SetAssociativity(UINT32 associativity) { ASSERTX(associativity
    == 1); }
UINT32 GetAssociativity(UINT32 associativity) { return 1; }

UINT32 Find(CACHE_TAG tag) { return(_tag == tag); }
VOID Replace(CACHE_TAG tag) { _tag = tag; }
VOID Flush() { _tag = 0; }
};

/*!
*  @brief Cache set with round robin replacement
*/
template <UINT32 MAX_ASSOCIATIVITY = 4>
class ROUND_ROBIN
{
private:
CACHE_TAG _tags[MAX_ASSOCIATIVITY];
UINT32 _tagsLastIndex;
UINT32 _nextReplaceIndex;

public:
ROUND_ROBIN(UINT32 associativity = MAX_ASSOCIATIVITY)
: _tagsLastIndex(associativity - 1)
{
ASSERTX(associativity <= MAX_ASSOCIATIVITY);
_nextReplaceIndex = _tagsLastIndex;

for (INT32 index = _tagsLastIndex; index >= 0; index--)
{
_tags[index] = CACHE_TAG(0);
}
}

VOID SetAssociativity(UINT32 associativity)
{
ASSERTX(associativity <= MAX_ASSOCIATIVITY);
_tagsLastIndex = associativity - 1;
_nextReplaceIndex = _tagsLastIndex;
}
UINT32 GetAssociativity(UINT32 associativity) { return _tagsLastIndex
    + 1; }

UINT32 Find(CACHE_TAG tag)
{
bool result = true;

for (INT32 index = _tagsLastIndex; index >= 0; index--)
{
// this is an ugly micro-optimization, but it does cause a
// tighter assembly loop for ARM that way ...
if(_tags[index] == tag) goto end;
}
result = false;

end: return result;
}

VOID Replace(CACHE_TAG tag)
{
// g++ -O3 too dumb to do CSE on following lines?!
const UINT32 index = _nextReplaceIndex;

_tags[index] = tag;
// condition typically faster than modulo
_nextReplaceIndex = (index == 0 ? _tagsLastIndex : index - 1);
}
VOID Flush()
{
for (INT32 index = _tagsLastIndex; index >= 0; index--)
{
_tags[index] = 0;
}
_nextReplaceIndex=_tagsLastIndex;
}
};

} // namespace CACHE_SET

namespace CACHE_ALLOC
{
typedef enum
{
STORE_ALLOCATE,
STORE_NO_ALLOCATE
} STORE_ALLOCATION;
}

/*!
*  @brief Generic cache base class; no allocate specialization, no
    cache set specialization
*/
class CACHE_BASE
{
public:
// types, constants
```

```
typedef enum
{
ACCESS_TYPE_LOAD,
ACCESS_TYPE_STORE,
ACCESS_TYPE_NUM
} ACCESS_TYPE;

protected:
static const UINT32 HIT_MISS_NUM = 2;
CACHE_STATS _access[ACCESS_TYPE_NUM][HIT_MISS_NUM];

private:
// input params
const std::string _name;
const UINT32 _cacheSize;
const UINT32 _lineSize;
const UINT32 _associativity;
UINT32 _numberOfFlushes;
UINT32 _numberOfResets;

// computed params
const UINT32 _lineShift;
const UINT32 _setIndexMask;

CACHE_STATS SumAccess(bool hit) const
{
CACHE_STATS sum = 0;

for (UINT32 accessType = 0; accessType < ACCESS_TYPE_NUM; accessType
    ++)
{
sum += _access[accessType][hit];
}

return sum;
}

protected:
UINT32 NumSets() const { return _setIndexMask + 1; }

public:
// constructors/destructors
CACHE_BASE(std::string name, UINT32 cacheSize, UINT32 lineSize,
    UINT32 associativity);

// accessors
UINT32 CacheSize() const { return _cacheSize; }
UINT32 LineSize() const { return _lineSize; }
```

```
UINT32 Associativity() const { return _associativity; }
//
CACHE_STATS Hits(ACCESS_TYPE accessType) const { return _access[
    accessType][true];}
CACHE_STATS Misses(ACCESS_TYPE accessType) const { return _access[
    accessType][false];}
CACHE_STATS Accesses(ACCESS_TYPE accessType) const { return Hits(
    accessType) + Misses(accessType);}
CACHE_STATS Hits() const { return SumAccess(true);}
CACHE_STATS Misses() const { return SumAccess(false);}
CACHE_STATS Accesses() const { return Hits() + Misses();}

CACHE_STATS Flushes() const { return _numberOfFlushes;}
CACHE_STATS Resets() const { return _numberOfResets;}

VOID SplitAddress(const ADDRINT addr, CACHE_TAG & tag, UINT32 &
    setIndex) const
{
tag = addr >> _lineShift;
setIndex = tag & _setIndexMask;
}

VOID SplitAddress(const ADDRINT addr, CACHE_TAG & tag, UINT32 &
    setIndex, UINT32 & lineIndex) const
{
const UINT32 lineMask = _lineSize - 1;
lineIndex = addr & lineMask;
SplitAddress(addr, tag, setIndex);
}

VOID IncFlushCounter()
{
_numberOfFlushes += 1;
}

VOID IncResetCounter()
{
_numberOfResets += 1;
}

std::ostream & StatsLong(std::ostream & out) const;
};

CACHE_BASE::CACHE_BASE(std::string name, UINT32 cacheSize, UINT32
    lineSize, UINT32 associativity)
: _name(name),
_cacheSize(cacheSize),
_lineSize(lineSize),
_associativity(associativity),
_lineShift(FloorLog2(lineSize)),
_setIndexMask((cacheSize / (associativity * lineSize)) - 1)
{

ASSERTX(IsPower2(_lineSize));
ASSERTX(IsPower2(_setIndexMask + 1));

for (UINT32 accessType = 0; accessType < ACCESS_TYPE_NUM; accessType
    ++)
{
_access[accessType][false] = 0;
_access[accessType][true] = 0;
}
}

/*!
*  @brief Stats output method
*/
std::ostream & CACHE_BASE::StatsLong(std::ostream & out) const
{
const UINT32 headerWidth = 19;
const UINT32 numberWidth = 10;

out << _name << ":" << std::endl;

for (UINT32 i = 0; i < ACCESS_TYPE_NUM; i++)
{
const ACCESS_TYPE accessType = ACCESS_TYPE(i);

std::string type(accessType == ACCESS_TYPE_LOAD ? "Load" : "Store");

out << StringString(type + " Hits:      ", headerWidth)
<< StringInt(Hits(accessType), numberWidth) << std::endl;
out << StringString(type + " Misses:    ", headerWidth)
<< StringInt(Misses(accessType), numberWidth) << std::endl;
out << StringString(type + " Accesses:  ", headerWidth)
<< StringInt(Accesses(accessType), numberWidth) << std::endl;
out << StringString(type + " Miss Rate: ", headerWidth)
<< StringFlt(100.0 * Misses(accessType) / Accesses(accessType), 2,
    numberWidth-1) << "%" << std::endl;
out << std::endl;
}

out << StringString("Total Hits:      ", headerWidth, ' ')
<< StringInt(Hits(), numberWidth) << std::endl;
out << StringString("Total Misses:    ", headerWidth, ' ')
<< StringInt(Misses(), numberWidth) << std::endl;
```

```
out << StringString("Total Accesses:  ", headerWidth, ' ')
<< StringInt(Accesses(), numberWidth) << std::endl;
out << StringString("Total Miss Rate: ", headerWidth, ' ')
<< StringFlt(100.0 * Misses() / Accesses(), 2, numberWidth-1) << "%"
    << std::endl;

out << StringString("Flushes:         ", headerWidth, ' ')
<< StringInt(Flushes(), numberWidth) << std::endl;
out << StringString("Stat Resets:     ", headerWidth, ' ')
<< StringInt(Resets(), numberWidth) << std::endl;

out << std::endl;

return out;
}

/// ostream operator for CACHE_BASE
std::ostream & operator<< (std::ostream & out, const CACHE_BASE &
    cacheBase)
{
return cacheBase.StatsLong(out);
}

/*!
*  @brief Templated cache class with specific cache set allocation
    policies
*
*  All that remains to be done here is allocate and deallocate the
    right
*  type of cache sets.
*/
template <class SET, UINT32 MAX_SETS, UINT32 STORE_ALLOCATION>
class CACHE : public CACHE_BASE
{
private:
SET _sets[MAX_SETS];

public:
// constructors/destructors
CACHE(std::string name, UINT32 cacheSize, UINT32 lineSize, UINT32
    associativity)
: CACHE_BASE(name, cacheSize, lineSize, associativity)
{
ASSERTX(NumSets() <= MAX_SETS);

for (UINT32 i = 0; i < NumSets(); i++)
{
_sets[i].SetAssociativity(associativity);
}
}

// modifiers
/// Cache access from addr to addr+size-1
bool Access(ADDRINT addr, UINT32 size, ACCESS_TYPE accessType);
/// Cache access at addr that does not span cache lines
bool AccessSingleLine(ADDRINT addr, ACCESS_TYPE accessType);
void Flush();
void ResetStats();
};

/*!
*  @return true if all accessed cache lines hit
*/

template <class SET, UINT32 MAX_SETS, UINT32 STORE_ALLOCATION>
bool CACHE<SET,MAX_SETS,STORE_ALLOCATION>::Access(ADDRINT addr,
    UINT32 size, ACCESS_TYPE accessType)
{
const ADDRINT highAddr = addr + size;
bool allHit = true;

const ADDRINT lineSize = LineSize();
const ADDRINT notLineMask = ~(lineSize - 1);
do
{
CACHE_TAG tag;
UINT32 setIndex;

SplitAddress(addr, tag, setIndex);

SET & set = _sets[setIndex];

bool localHit = set.Find(tag);
allHit &= localHit;

// on miss, loads always allocate, stores optionally
if ( (! localHit) && (accessType == ACCESS_TYPE_LOAD ||
    STORE_ALLOCATION == CACHE_ALLOC::STORE_ALLOCATE))
{
set.Replace(tag);
}

addr = (addr & notLineMask) + lineSize; // start of next cache line
}
while (addr < highAddr);

```

```
_access[accessType][allHit]++;

return allHit;
}

/*!
*  @return true if accessed cache line hits
*/
template <class SET, UINT32 MAX_SETS, UINT32 STORE_ALLOCATION>
bool CACHE<SET,MAX_SETS,STORE_ALLOCATION>::AccessSingleLine(ADDRINT
    addr, ACCESS_TYPE accessType)
{
CACHE_TAG tag;
UINT32 setIndex;

SplitAddress(addr, tag, setIndex);

SET & set = _sets[setIndex];

bool hit = set.Find(tag);

// on miss, loads always allocate, stores optionally
if ( (! hit) && (accessType == ACCESS_TYPE_LOAD || STORE_ALLOCATION
    == CACHE_ALLOC::STORE_ALLOCATE))
{
set.Replace(tag);
}

_access[accessType][hit]++;

return hit;
}
/*!
*  @return true if accessed cache line hits
*/
template <class SET, UINT32 MAX_SETS, UINT32 STORE_ALLOCATION>
void CACHE<SET,MAX_SETS,STORE_ALLOCATION>::Flush()
{
for (INT32 index = NumSets(); index >= 0; index--) {
SET & set = _sets[index];
set.Flush();
}
IncFlushCounter();
}

template <class SET, UINT32 MAX_SETS, UINT32 STORE_ALLOCATION>
void CACHE<SET,MAX_SETS,STORE_ALLOCATION>::ResetStats()
{
```

```
for (UINT32 accessType = 0; accessType < ACCESS_TYPE_NUM; accessType
    ++)
{
_access[accessType][false] = 0;
_access[accessType][true] = 0;
}
IncResetCounter();
}


// define shortcuts
#define CACHE_DIRECT_MAPPED(MAX_SETS, ALLOCATION) CACHE<CACHE_SET::
    DIRECT_MAPPED, MAX_SETS, ALLOCATION>
#define CACHE_ROUND_ROBIN(MAX_SETS, MAX_ASSOCIATIVITY, ALLOCATION)
    CACHE<CACHE_SET::ROUND_ROBIN<MAX_ASSOCIATIVITY>, MAX_SETS,
    ALLOCATION>

#endif // PIN_CACHE_H
```

boost/preprocessor/iteration/local.hpp

```
# /* **************************************************************
#  *     (C) Copyright Paul Mensonides 2002.
#  *     Distributed under the Boost Software License, Version 1.0.
#  *     (See accompanying file LICENSE_1_0.txt or copy at
#  *     http://www.boost.org/LICENSE_1_0.txt)
#  ************************************************************** */
#
# /* See http://www.boost.org for most recent version. */
#
# ifndef BOOST_PREPROCESSOR_ITERATION_LOCAL_HPP
# define BOOST_PREPROCESSOR_ITERATION_LOCAL_HPP
#
# include <boost/preprocessor/config/config.hpp>
# include <boost/preprocessor/slot/slot.hpp>
# include <boost/preprocessor/tuple/elem.hpp>
#
# /* BOOST_PP_LOCAL_ITERATE */
#
# define BOOST_PP_LOCAL_ITERATE() <boost/preprocessor/iteration/
    detail/local.hpp>
#
```

```
# define BOOST_PP_LOCAL_C(n) (BOOST_PP_LOCAL_S) <= n && (
    BOOST_PP_LOCAL_F) >= n
# define BOOST_PP_LOCAL_R(n) (BOOST_PP_LOCAL_F) <= n && (
    BOOST_PP_LOCAL_S) >= n
#
# endif
```

B APPENDIX

Simulation output

The Simulation Output consists of two discrete output files: Pintool's .out file and DRAMSim2 .log file.

Pintool .out file

These files contain information about Ram accesses and their percentage per Window, the Local and Disaggregated Clock cycles, the Total and Total Instrumented Instructions, the Local and Disaggregated Cycles per Instruction (CPI) and finaly a complete Cache report. The Cache report contains information about Load/Store/Total Hits, Misses, Accesses, Miss Rate for each Cache Level separately. A simple example is following:

```
Ram Accesses: 629850
Ram Percentage: 0.31%
 Simulation number: 1 of 10
==============================

Ram Accesses: 634720
Ram Percentage: 0.32%
 Simulation number: 2 of 10
```

```
==============================

Ram Accesses: 635207
Ram Percentage: 0.32%
 Simulation number: 3 of 10
==============================

Ram Accesses: 638265
Ram Percentage: 0.32%
 Simulation number: 4 of 10
==============================

Ram Accesses: 637306
Ram Percentage: 0.32%
 Simulation number: 5 of 10
==============================

Ram Accesses: 635036
Ram Percentage: 0.32%
 Simulation number: 6 of 10
==============================

Ram Accesses: 632138
Ram Percentage: 0.32%
 Simulation number: 7 of 10
==============================

Ram Accesses: 637267
Ram Percentage: 0.32%
 Simulation number: 8 of 10
```

```
==============================

Ram Accesses: 636057
Ram Percentage: 0.32%
 Simulation number: 9 of 10
==============================

Ram Accesses: 640913
Ram Percentage: 0.32%
 Simulation number: 10 of 10
==============================

 Local Clock: 992749854
 Disaggregated Clock: 12574239309

 Total Instructions: 4400000169
 Total Instrumented Instructions: 2000000000

 0.496 cyles per Instruction (L)
 6.29 cyles per Instruction (D)

 Simulation started at (04:35:14 | 2017-03-03) and ended at (06:41:00
    | 2017-03-03)

DTLB 0:
 Load Hits:        163066592
 Load Misses:             45
 Load Accesses:    163066637
 Load Miss Rate:       0.00%

```

```
Store Hits:                 0
Store Misses:               0
Store Accesses:             0
Store Miss Rate:        -nan%

Total Hits:         163066592
Total Misses:              45
Total Accesses:     163066637
Total Miss Rate:        0.00%
Flushes:                    0
Stat Resets:                0

L1 Data Cache 0:
Load Hits:         102827682
Load Misses:         2021252
Load Accesses:     104848934
Load Miss Rate:        1.93%

Store Hits:          53080510
Store Misses:         5137193
Store Accesses:      58217703
Store Miss Rate:        8.82%

Total Hits:         155908192
Total Misses:         7158445
Total Accesses:     163066637
Total Miss Rate:        4.39%
Flushes:                    0
Stat Resets:                0

```

```
L2 Data Cache 0:
Load Hits:            854045
Load Misses:         1167207
Load Accesses:       2021252
Load Miss Rate:       57.75%

 Store Hits:            4705229
 Store Misses:           431964
 Store Accesses:        5137193
 Store Miss Rate:         8.41%

 Total Hits:            5559274
 Total Misses:          1599171
 Total Accesses:        7158445
 Total Miss Rate:        22.34%
 Flushes:                     0
 Stat Resets:                 0

DTLB 1:
Load Hits:         158597676
Load Misses:              45
Load Accesses:     158597721
Load Miss Rate:        0.00%

 Store Hits:                  0
 Store Misses:                0
 Store Accesses:              0
 Store Miss Rate:         -nan%

 Total Hits:          158597676
Total Misses:             45
Total Accesses:    158597721
Total Miss Rate:       0.00%
Flushes:                   0
Stat Resets:               0

L1 Data Cache 1:
Load Hits:          99947603
Load Misses:         1975900
Load Accesses:     101923503
Load Miss Rate:        1.94%

Store Hits:         51726528
Store Misses:        4947690
Store Accesses:     56674218
Store Miss Rate:       8.73%

Total Hits:        151674131
Total Misses:        6923590
Total Accesses:    158597721
Total Miss Rate:       4.37%
Flushes:                   0
Stat Resets:               0

L2 Data Cache 1:
Load Hits:            830255
Load Misses:         1145645
Load Accesses:       1975900
Load Miss Rate:       57.98%

```

```
Store Hits:          4525207
Store Misses:         422483
Store Accesses:      4947690
Store Miss Rate:       8.54%

Total Hits:          5355462
Total Misses:        1568128
Total Accesses:      6923590
Total Miss Rate:      22.65%
Flushes:                   0
Stat Resets:               0

DTLB 2:
Load Hits:         159593926
Load Misses:              43
Load Accesses:     159593969
Load Miss Rate:        0.00%

Store Hits:                0
Store Misses:              0
Store Accesses:            0
Store Miss Rate:       -nan%

Total Hits:        159593926
Total Misses:             43
Total Accesses:    159593969
Total Miss Rate:       0.00%
Flushes:                   0
Stat Resets:               0
```

```
L1 Data Cache 2:
Load Hits:        100668087
Load Misses:        1930028
Load Accesses:    102598115
Load Miss Rate:       1.88%

Store Hits:        52171348
Store Misses:       4824506
Store Accesses:    56995854
Store Miss Rate:      8.46%

Total Hits:       152839435
Total Misses:       6754534
Total Accesses:   159593969
Total Miss Rate:      4.23%
Flushes:                  0
Stat Resets:              0

L2 Data Cache 2:
Load Hits:           778062
Load Misses:        1151966
Load Accesses:      1930028
Load Miss Rate:      59.69%

Store Hits:         4399951
Store Misses:        424555
Store Accesses:     4824506
Store Miss Rate:      8.80%

Total Hits:         5178013
```

```
 Total Misses:        1576521
 Total Accesses:      6754534
 Total Miss Rate:      23.34%
 Flushes:                   0
 Stat Resets:               0

DTLB 3:
Load Hits:         164599458
Load Misses:              44
Load Accesses:     164599502
Load Miss Rate:        0.00%

 Store Hits:                0
 Store Misses:              0
 Store Accesses:            0
 Store Miss Rate:       -nan%

 Total Hits:        164599458
 Total Misses:             44
 Total Accesses:    164599502
 Total Miss Rate:       0.00%
 Flushes:                   0
 Stat Resets:               0

L1 Data Cache 3:
Load Hits:         103791118
Load Misses:         2067925
Load Accesses:     105859043
Load Miss Rate:        1.95%

```

```
Store Hits:           53658600
Store Misses:          5081859
Store Accesses:       58740459
Store Miss Rate:         8.65%

Total Hits:          157449718
Total Misses:          7149784
Total Accesses:      164599502
Total Miss Rate:         4.34%
Flushes:                     0
Stat Resets:                 0

L2 Data Cache 3:
Load Hits:              885205
Load Misses:           1182720
Load Accesses:         2067925
Load Miss Rate:         57.19%

Store Hits:            4651640
Store Misses:           430219
Store Accesses:        5081859
Store Miss Rate:         8.47%

Total Hits:            5536845
Total Misses:          1612939
Total Accesses:        7149784
Total Miss Rate:        22.56%
Flushes:                     0
```

```
Stat Resets:                0
```

Listing B.1: Pintool.out example

DRAMSim2 .log file

These files contain a DRAM report per `EPOCH`. The report presents statistical results about the current DRAM state per Rank. For each Rank, at the begining are presented the Total Return Trasactions and an average Bandwidth. Then are listed the Reads, Writes, Latency and Bandwidth (per Bank) and Power Data for that Rank. At the end of a .log file is presented a complete Latency Histogram for all Ranks. A simple example is following:

```
 | Benchmark: DataCaching | CPU: Cortex-A53 | Scenario: 25-75 | Step:
    7 |
=====================================================
============== Printing Statistics [id:0]==============
Total Return Transactions : 0 (0 bytes) aggregate average bandwidth
    0.000MB/s
-Rank   0 :
-Reads  : 0 (0 bytes)
-Writes : 0 (0 bytes)
-Bandwidth / Latency  (Bank 0): 0.000 MB/s    -nan   s
-Bandwidth / Latency  (Bank 1): 0.000 MB/s    -nan   s
-Bandwidth / Latency  (Bank 2): 0.000 MB/s    -nan   s
-Bandwidth / Latency  (Bank 3): 0.000 MB/s    -nan   s
-Bandwidth / Latency  (Bank 4): 0.000 MB/s    -nan   s
-Bandwidth / Latency  (Bank 5): 0.000 MB/s    -nan   s
-Bandwidth / Latency  (Bank 6): 0.000 MB/s    -nan   s
```

```
-Bandwidth / Latency  (Bank 7): 0.000 MB/s    -nan   s
-Bandwidth / Latency  (Bank 8): 0.000 MB/s    -nan   s
-Bandwidth / Latency  (Bank 9): 0.000 MB/s    -nan   s
-Bandwidth / Latency  (Bank 10): 0.000 MB/s   -nan   s
-Bandwidth / Latency  (Bank 11): 0.000 MB/s   -nan   s
-Bandwidth / Latency  (Bank 12): 0.000 MB/s   -nan   s
-Bandwidth / Latency  (Bank 13): 0.000 MB/s   -nan   s
-Bandwidth / Latency  (Bank 14): 0.000 MB/s   -nan   s
-Bandwidth / Latency  (Bank 15): 0.000 MB/s   -nan   s
== Power Data for Rank        0
Average Power (mW)      : 0.000
-Background (mW)      : 0.000
-Act/Pre    (mW)      : 0.000
-Burst      (mW)      : 0.000
-Refresh    (mW)      : 0.000

== Pending Transactions : 0 (0)==
==== Channel [0] ====
| Benchmark: DataCaching | CPU: Cortex-A53 | Scenario: 25-75 | Step:
    7 |
============================================================
=============== Printing Statistics [id:0]===============
Total Return Transactions : 1636284 (104722176 bytes) aggregate
    average bandwidth 118.353MB/s
-Rank   0 :
-Reads  : 1196027 (76545728 bytes)
-Writes : 440257 (28176448 bytes)
-Bandwidth / Latency  (Bank 0): 7.488 MB/s    34040.604   s
-Bandwidth / Latency  (Bank 1): 7.436 MB/s    33998.760   s
-Bandwidth / Latency  (Bank 2): 7.410 MB/s    34029.888   s
```

```
-Bandwidth / Latency  (Bank 3): 7.356 MB/s    33996.971   s
-Bandwidth / Latency  (Bank 4): 7.343 MB/s    34047.424   s
-Bandwidth / Latency  (Bank 5): 7.287 MB/s    34035.362   s
-Bandwidth / Latency  (Bank 6): 7.350 MB/s    34057.701   s
-Bandwidth / Latency  (Bank 7): 7.413 MB/s    34086.468   s
-Bandwidth / Latency  (Bank 8): 7.364 MB/s    34009.594   s
-Bandwidth / Latency  (Bank 9): 7.425 MB/s    34010.433   s
-Bandwidth / Latency  (Bank 10): 7.513 MB/s   34044.460   s
-Bandwidth / Latency  (Bank 11): 7.411 MB/s   34054.859   s
-Bandwidth / Latency  (Bank 12): 7.378 MB/s   34092.583   s
-Bandwidth / Latency  (Bank 13): 7.337 MB/s   34075.726   s
-Bandwidth / Latency  (Bank 14): 7.379 MB/s   34171.051   s
-Bandwidth / Latency  (Bank 15): 7.464 MB/s   34069.420   s
== Power Data for Rank        0
 Average Power (mW)     : 1022.263
-Background (mW)     : 983.599
-Act/Pre    (mW)     : 13.160
-Burst      (mW)     : 23.494
-Refresh    (mW)     : 2.009
---  Latency list (28)
 [lat] : #
 [20-29] : 205
 [30-39] : 1120626
 [40-49] : 39903
 [50-59] : 6047
 [60-69] : 6657
 [70-79] : 2554
 [80-89] : 4143
 [90-99] : 1732
 [100-109] : 1427
```

```
[110-119] : 1385
[120-129] : 1295
[130-139] : 1152
[140-149] : 1150
[150-159] : 1766
[160-169] : 1304
[170-179] : 1290
[180-189] : 1412
[190-199] : 1195
[200-209] : 274
[210-219] : 108
[220-229] : 233
[230-239] : 48
[240-249] : 102
[250-259] : 13
[260-269] : 3
[270-279] : 1
[290-299] : 1
[310-319] : 1

== Pending Transactions : 0 (992751922)==
//// Channel [0] ////
```

Listing B.2: DRAMSim2.out example

Bibliography

[1] Robert Barnes. "Best Practices for Benchmarking and Performance Analysis in the Cloud (ENT305)". In: Amazon Web Services, 2013. URL: `http://www.slideshare.net/AmazonWebServices/best-practices-for-benchmarking-and-performance-analysis-in-the-cloud-ent305-aws-reinvent-2013-28437412?qid=15d7ec92-5db7-48d5-a764-01290a9021c4`.

[2] Andrew R Bernat and Barton P Miller. "Anywhere, any-time binary instrumentation". In: *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*. ACM. 2011, pp. 9–16.

[3] Derek Bruening. *QZ: Dynamorio: Dynamic instrumentation tool platform.*

[4] K Chandrasekaran. *Essentials of cloud computing.* CRC Press, 2014.

[5] Gal Diskin. "Binary Instrumentation for Security Professionals". In: *Intel. Blackhat USA* (2011).

[6] Robert Demming & Daniel J. Duffy. *Introduction to the Boost C++ Libraries; Volume I - Foundations.* Datasim Education BV, 2010. ISBN: 9789491028014.

[7] Karthi Duraisamy and Goutam Das. "WMRD net: An Optical Data Center Interconnect". In: *Optical Fiber Communication Conference/National Fiber Optic Engineers Conference 2013.* Optical Society of America, 2013,

OTu3H.3. DOI: 10.1364/OFC.2013.OTu3H.3. URL: http://www.osapublishing.org/abstract.cfm?URI=OFC-2013-OTu3H.3.

[8] D. Evdokimov. *Light and dark side of code instrumentation.* 2013.

[9] Suhaib A Fahmy, Kizheppatt Vipin, and Shanker Shreejith. "Virtualized FPGA accelerators for efficient cloud computing". In: *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on.* IEEE. 2015, pp. 430–435.

[10] Michael Ferdman et al. "Clearing the clouds: a study of emerging scale-out workloads on modern hardware". In: *ACM SIGPLAN Notices.* Vol. 47. 4. ACM. 2012, pp. 37–48.

[11] Robert E Filman and Klaus Havelund. *Source-code instrumentation and quantification of events.* 2002.

[12] Sangjin Han et al. "Network support for resource disaggregation in next-generation datacenters". In: *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks.* ACM. 2013, p. 10.

[13] Aamer Jaleel et al. "CMPSim: A Pin-based on-the-fly multi-core cache simulator". In: *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA.* 2008, pp. 28–36.

[14] D. I. Jeon and K. S. Chung. "CasHMC: A Cycle-Accurate Simulator for Hybrid Memory Cube". In: *IEEE Computer Architecture Letters* 16.1 (Jan. 2017), pp. 10–13. ISSN: 1556-6056. DOI: 10.1109/LCA.2016.2600601.

[15] Johnls. *Dynamic vs. Static Instrumentation @ONLINE.* Nov. 2006. URL: `https://blogs.msdn.microsoft.com/johnls/2006/11/15/dynamic-vs-static-instrumentation/`.

[16] K. Katrinis et al. "On interconnecting and orchestrating components in disaggregated data centers: The dReDBox project vision". In: *2016 European Conference on Networks and Communications (EuCNC).* June 2016, pp. 235–239. DOI: `10.1109/EuCNC.2016.7561039`.

[17] K. Katrinis et al. "Rack-scale disaggregated cloud data centers: The dReDBox project vision". In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE).* Mar. 2016, pp. 690–695.

[18] Y. Kim, W. Yang, and O. Mutlu. "Ramulator: A Fast and Extensible DRAM Simulator". In: *IEEE Computer Architecture Letters* 15.1 (Jan. 2016), pp. 45–49. ISSN: 1556-6056. DOI: `10.1109/LCA.2015.2414456`.

[19] J. D. Leidel and Y. Chen. "HMC-Sim-2.0: A Simulation Platform for Exploring Custom Memory Cube Operations". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* May 2016, pp. 621–630. DOI: `10.1109/IPDPSW.2016.43`.

[20] Kevin Lim et al. "Disaggregated memory for expansion and sharing in blade servers". In: *ACM SIGARCH Computer Architecture News.* Vol. 37. 3. ACM. 2009, pp. 267–278.

[21] Kevin Lim et al. "System-level implications of disaggregated memory". In: *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on.* IEEE. 2012, pp. 1–12.

[22] Chi-Keung Luk et al. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *Acm sigplan notices*. Vol. 40. 6. ACM. 2005, pp. 190–200.

[23] Hugo Meyer et al. “Disaggregated Computing. An Evaluation of Current Trends for Datacentres”. In: *Procedia Computer Science* 108 (2017), pp. 685–694.

[24] Nicholas Nethercote and Julian Seward. “Valgrind: A program supervision framework”. In: *Electronic notes in theoretical computer science* 89.2 (2003), pp. 44–66.

[25] Tapti Palit, Yongming Shen, and Michael Ferdman. “Demystifying cloud benchmarking”. In: *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*. IEEE. 2016, pp. 122–132.

[26] Orion Papadakis. *Memory System Evaluation of Disaggregated High Performance Parallel Systems*. 2017.

[27] Vijay Janapa Reddi et al. “PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education”. In: *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*. WCAE ’04. Munich, Germany: ACM, 2004. ISBN: 978-1-4503-4733-4. DOI: `10.1145/1275571.1275600`. URL: `http://doi.acm.org/10.1145/1275571.1275600`.

[28] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. “DRAMSim2: A cycle accurate memory system simulator”. In: *IEEE Computer Architecture Letters* 10.1 (2011), pp. 16–19.

[29] Open Source. “Dyninst: An application program interface (api) for runtime code generation”. In: *Online, http://www. dyninst. org* ().

[30] Gregory T. Sullivan et al. “Dynamic Native Optimization of Interpreters”. In: *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*. IVME ’03. San Diego, California: ACM, 2003, pp. 50–57. ISBN: 1-58113-655-2. DOI: `10.1145/858570.858576`. URL: `http://doi.acm.org/10.1145/858570.858576`.

[31] Gang-Ryung Uh et al. “Analyzing dynamic binary instrumentation overhead”. In: *WBIA Workshop at ASPLOS*. 2006.

[32] Georgios S. Zervas et al. “Disaggregated Compute, Memory and Network Systems: A New Era for Optical Data Centre Architectures”. In: *Optical Fiber Communication Conference*. Optical Society of America, 2017, W3D.4. DOI: `10.1364/OFC.2017.W3D.4`. URL: `http://www.osapublishing.org/abstract.cfm?URI=OFC-2017-W3D.4`.