Technical University of Crete
School of Electronics and Computer Engineering
Department of Electronics and Computer Engineering

# Memory System Evaluation of Disaggregated High Performance Parallel Systems

Orion Papadakis

Thesis Commitee:
Prof. Dionisios N. Pneumatikatos (ECE) (Supervisor)
Prof. Vasilis Samoladas (ECE)
Dr. Dimitris Theodoropoulos (ECE)

Submitted in part fulfilment of the requirements for the diploma of
Electronics and Computer Engineering, April 2017

## Abstract

Supercomputers or High Performance Computers (HPC), traditionally play a significant role either in the Computer Architecture scientific field, or in the Computer Science, due to their usage in manner computing processes, scientific research and applications. Consequently, the study around them, as well as, the Memory System performance and size study is necessary about their further evolve, due to the traditional bottleneck between Memory and CPU speed (memory gap). Large resources inefficiencies (mostly in Memory) as well as, significant power consumption regarding to the current Cloud Data Center structure, have been observed. Their mainboard-oriented monolithic structure fails to operate in an optimal way with the hardware, corresponding to the modern application needs. Larger Data Center are being built, in response to that problem, a strategy which leads to even more power consumption. The nowadays research about Disaggregated Architecture Systems, study those problems. It aims to change the traditional mainboard-organized Data Center structure by proposing a more flexible and software-controlled one, organized around Pooled Disaggregated Resources.

The current diploma thesis is part of the DiMEM Simulator, a modular execution-driven Disaggregated Memory Simulation tool study and implementation. That tool approximately tries to depict the Disaggregated Memory System behaviour using HPC workload. The DiMEM Simulator couples the Intel PIN framework with DRAMSim2 Memory Simulator, where that thesis also focuses. The main study object are the DRAMs, the Memory Simulation methods, the Disaggregated Memory Simulation implementation, as well as the parameters experimentation. The presented results show the approximated Disaggregated Memory System behaviour.

## Περίληψη

Οι Υπερυπολογιστές ή Υπολογιστές Υψηλών Επιδόσεων (High Performance Computers) διαχρονικά, κατέχουν σημαντικό ρόλο στο πεδίο της Αρχιτεκτονικής Υπολογιστών αλλά και γενικότερα στην Επιστήμη Υπολογιστών καθώς χρησιμοποιούνται σε ιδιαίτερα απαιτητικές υπολογιστικά εργασίες και βαρύνουσας σημασίας επιστημονική έρευνα και εφαρμογές. Συνεπώς η μελέτη τους, και πιο συγκεκριμένα, η μελέτη των συστημάτων μνήμης τους όσον αφορά τις επιδόσεις και το μέγεθός τους είναι αναγκαία για την περαιτέρω ανάπτυξη των συστημάτων αυτών καθώς παραδοσιακά το (memory gap-wall) παίζει επιβραδυντικό ρόλο στις ήδη υψηλές επιδόσεις των επεξεργαστών. Όσον αφορά την υπάρχουσα δομή των μεγάλων Cloud Data Centers παρατηρείται σημαντική σπατάλη πόρων (κυρίως μνήμης) και υψηλή κατανάλωση ενέργειας. Η μονολιθική τους δομή με επίκεντρο το mainboard δεν αξιοποιεί βέλτιστα το hardware για τις ανάγκες των εφαρμογών και εμφανίζονται ανεπάρκειες. Έτσι, προκειμένου να επιστρατευτούν οι αναγκαίοι πόροι, χτίζονται πιο μεγάλα Data Centers, πρακτική που οδηγεί μεταξύ άλλων και στην ακόμα υψηλότερη κατανάλωση ενέργειας. Στην κατεύθυνση αυτή, βρίσκονται υπο έρευνα και μελέτη Συστήματα Απομακρυσμένης/Επιμερισμένης Αρχιτεκτονικής (Disaggregated Architecture Systems). Η Αρχιτεκτονική αυτή στοχεύει να αλλάξει τον παραδοσιακό τρόπο οργάνωσης ενός Data Center, προτείνοντας την μετακόμιση από την ενοποίηση γυρω από το mainboard σε μια πιο ευέλικτη και μεταβαλόμενη από το software ενοποίηση γύρω από blocks, τις Επιμερισμένες Δεξαμενές Πόρων (Pooled Disaggregated Resources).

Η διπλωματική αυτή είναι κομμάτι της μελέτης και ανάπτυξης ένα ενοποιημένου (modular) εργαλείου προσομοίωσης μνήμης «οδηγούμενο» από την εκτέλεση ενός προγράμματος (execution driven) με σκοπό να αποτυπωθεί προσεγγιστικά η συμπεριφορά του τυπικού HPC φόρτου εργασίας σε συνθήκες Μνήμης Επιμερισμένης Αρχιτεκτονικής. Το εργαλείο συνενώνει το Intel Pin Framework με τον προσομοιωτή Μνήμης DRAMSim2 όπου και επικεντρώνεται η διπλωματική. Αντικείμενο μελέτης είναι οι μνήμες DRAM καθώς και οι μέθοδοι προσομοίωσής τους, η υλοποίηση της προσομοίωσης Μνήμης Επιμερισμένης Αρχιτεκτονικής, και ο πειραματισμός με τις διάφορες παραμέτρους. Τα αποτελέσματα που παρουσιάζονται αποτυπώνουν προσεγγιστικά τη γενικότερη συμπεριφορά ενός συστήματος Μνήμης Επιμερισμένης Αρχιτεκτονικής.

# Acknowledgements

First and foremost, I would like to thank my advisor, Professor Dionisios N. Pnevmatikatos for his motivation, immense knowledge and for giving me the opportunity to work on this very interesting topic.

Also I would like to thank Dr.Dimitris Theodoropoulos for his technical support during this thesis.

I would like to thank my colleague Andreas Andronikakis, that without our productive collaboration, our whole work and study would have never been accomplished.

Moreover, I would like to thank all those friends who during my studies were always supportive and created for me millions of beautiful memories.

Finally, most of all, I would like to thank my family for their enormous help, understanding and supporting me in all levels.

# Contents

# List of Tables

# List of Figures

# Listings

# Chapter 1

# Introduction

*High-performance computing* (HPC) or Supercomputers is the use of parallel processing for running advanced application programs efficiently, reliably and quickly. Performance of those systems is measured above a *teraflop (*$10^{12}$*)* floating point operations per second. Some of them are near or at the current highest performance. For instance, the Chinese supercomputer *Sunway TaihuLight*, which is the first in the TOP500 list (June 2016) as the fastest supercomputer in the world, reaches *93 petaflops (*$10^{15}$*)* on the LINPACK benchmarks. Since HPC are commonly used by academic institutes, researchers and engineers, their study constitutes a crucial and traditionally continuous researching field of Computer Architecture.

*Disaggregated Architecture Systems* are under research by the H2020 EU project dRedBox[10], that aims to break the system resources and scalability limitations, which lead to inefficiencies, sub-optimal resource availability and unexploited spare resources in current datacenters. Furthermore, it introduces a new Rack-Scale Architecture that will not require memory or accelerator to be co-located with a processor in the same node. System resources are aimed to be connected via multiple networks. High-speed and low-latency electrical network will be used for intra - tray data access in memory bricks. High-speed low-latency optical network will be used for inter - tray data access in memory bricks. The current thesis is trying to do an experimental evaluation of that memory inter-connection.

Before a particular design "committed to silicon", we need a way to *evaluate* the performance of

design alternatives quickly, accurately and costlessly. An initial assessment and exploration for the whole set of design parameters is absolutely necessary, in order to be reliable for further research and development. System Simulation is a good option for that purpose because it is cheap and easy to develop compared with a real system implementation.

The goal of the current diploma thesis is the development of a Disaggregated Memory System Simulator and its use for the evaluation of Disaggregated High Performance Parallel Systems. The DIMEM Simulator has to operate by passing all the Main Memory CPU requests to a Disaggregated Memory System model. Furthermore it must aggregates statistical results, such as CLK, Bandwidth, in order, to make the user able to evaluate the Simulation.

In Memory Simulation a software system that models and simulates Memory is needed, together with a suitable method to model the traffic of the System. The work of DIMEM Simulator is based on the DRAMSim2, a Cycle Accurate Memory System Simulator (for modelling and simulation). This way implementation of the traffic generation and passing methods to DRAMSim2 are the missing pieces of the puzzle. Intel's PIN dynamic binary instrumentation framework was chosen as the development means of a dynamic program analysis tool for memory tracing and preparing the simulation.

The tool which was created using Intel's Pin (Pintool) combines two main goals: 1)Binary Executable Memory Tracing, 2) Memory Simulation. These two goals were expanded in two discrete but interrelated Diploma Thesis. The first one has been covered by Andreas Andronikakis [9], and the second is the object of the current thesis. These two goals compose the whole Disaggregated Memory System Simulator (DiMEM), which can be used for evaluation of any type of Disaggregated Memory System.

The current 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 will be described in detail in chapter 3 are:

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” and Statistical processing of the results.

In the implementation chapter code parts are presented to help readers understanding.

High-Performance Parallel Systems are the object of Disaggregated Memory System Evaluation process which will be described in chapter 4. Segments of both Splash-3 and PARSEC Benchmark Suites have been used for evaluation.

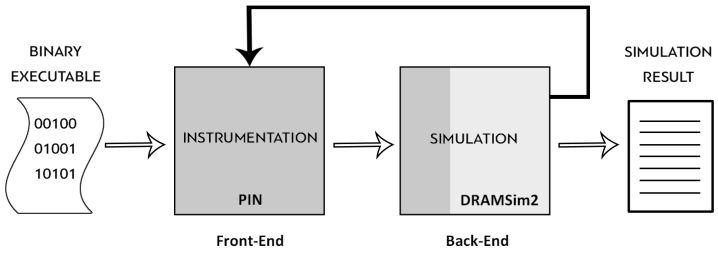


Figure 1.1: Whole System Abstract Figure

# Chapter 2

# Background And Related Work

## 2.1 Background

### 2.1.1 DRAM Technology Overview

The random-access memory technology (RAM) is used for the Main Memory implementation of a Computer System. Main Memory is responsible for the majority of the needed information required during a program execution. Despite the fact that it is commonly conceived as a *box* that receives CPU requests and responses with data, Main Memory has a structure which is critical for its performance. These structural characteristics are about to be described for the purpose of understanding the evaluation approach.

#### The Basic Circuit

RAM that uses a single *transistor-capacitor pair* for each bit is called a dynamic random-access memory or DRAM. The circuit is dynamic because the capacitors storing electrons are not perfect devices, and their eventual leakage requires that, to retain information stored there, each capacitor in the DRAM must be periodically refreshed (read and rewritten).[1]

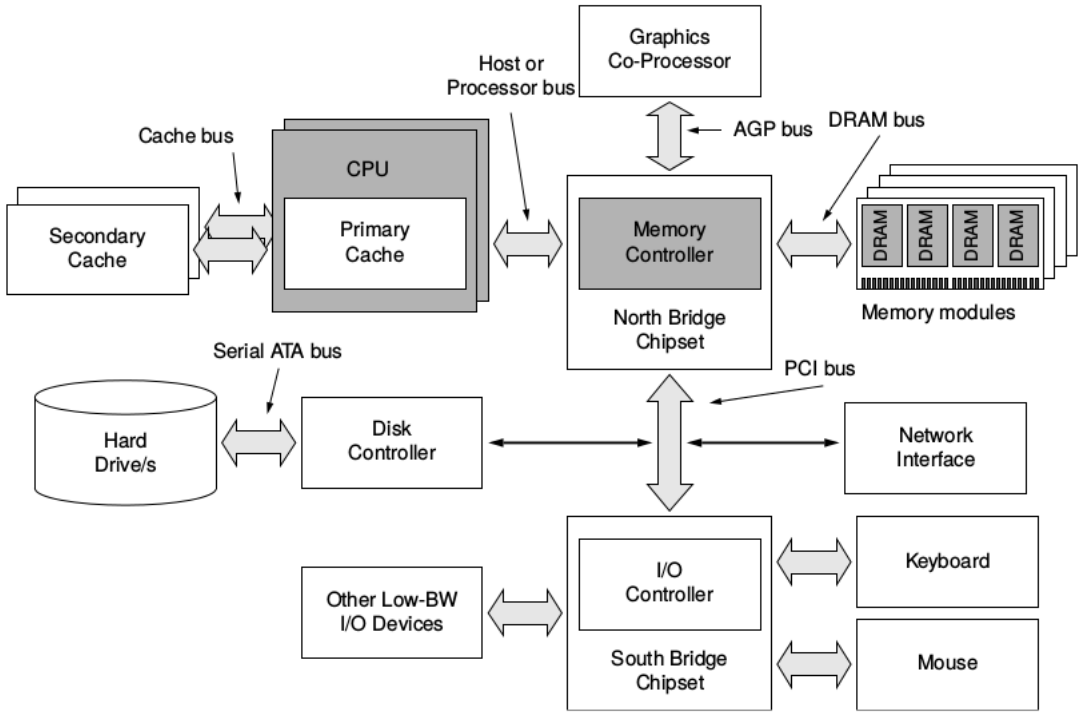


Figure 2.1: A typical PC organization and the DRAM subsystem as one part of a complex whole.

### Memory Arrays

Each DRAM device/chip contains one or more *memory arrays*, rectangular grinds of storage cells with each cell holding one bit of data. The arrays are organized into rows and columns. The Memory Controller (MC) uses the rows and columns (row address and column address) to access a specific storage cell into the DRAM chip. In the case of more than one memory array, a DRAM chip works in several different ways: in unison, completely independently or somewhere in the middle. If the memory arrays are designed to act in unison, they work as a unit, and the memory chip transmits or receives a number of bits equal to the number of arrays (Device Density or Device Width) each time the memory controller accesses the DRAM. For example, an x4 DRAM indicates that the DRAM device has four memory arrays and that a column width is 4 bits (each column read or write transmits 4 bits of data). In an x4 DRAM part, four arrays each read 1 data bit in unison, and the part sends out 4 bits of data each time the MC makes a column read request.

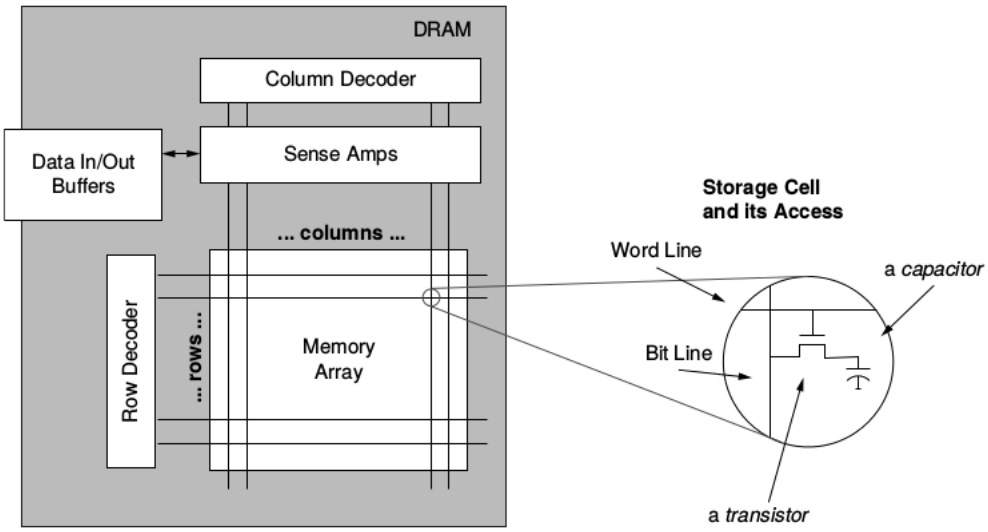


Figure 2.2: Basic organization of DRAM internals.

### Banks and Devices

A set of memory arrays that operates independently from other sets is referred to as a *Bank*. Each Bank is independent and can be activated, precharged, read out, etc. at the same time that other Banks (on the same or on other DRAM devices) are being activated, precharged, etc. The use of multiple independent Banks of memory has been a common practice in computer design since DRAMs were invented. One or more Banks which work in unison are organized and compose a DRAM device. As a result, interleaving multiple memory banks has been a popular method, used to achieve high-bandwidth memory buses using low-bandwidth devices.

Despite the existent confusion, among the bibliography, about the clear definition of word "Bank", it is currently used by DRAM device manufacturers to describe the number of independent *DRAM arrays* within a DRAM device [6]. For example a *16 Banks* ***x4*** DRAM Device contains 16 *independent* ***quadruple*** Arrays. Each quadruple Array acts independently from the 15 others, but the 4 internally contained Arrays act in parallel.

### Ranks

As a system can have multiple DIMMs, each of which can be perceived as an independent Bank, and the DRAM devices can implement internally multiple independent Banks, the word "Rank" was introduced to distinguish DIMM-level independent operation. Each Rank is a set of DRAM devices that operates in unison, and internally each of these DRAM Devices implements one or more independent Banks. JEDEC standard defines that a Rank must have 64 bits output bus so the number of DRAM Devices in a Rank is equal to 64/Device Width. Each DIMM can contain exactly one or two Ranks.

### DIMM/Channel

The Rank or Ranks compose a DIMM, or in other words a Channel. A computer's Memory System can have a single Channel, or multiple Channels. As an overview, a Channel is the collection of all DRAM Devices that share a common physical link (command, address, data buses) to the processor, and , thus, although a set of DIMMs in the Channel receives the same command, only one Rank replies. A Multi-channel system can be further divided into (1) multiple dependent (lockstep) Channels: single Memory Controller with ganged Channels to provide a wider interface; and (2) multiple independent Channels (each with its own Controller).

The number of Banks is also used as a way to characterize a whole DIMM, without considering as the total *physical* number of Banks that a whole DIMM may contains. The physical number of contained Banks = Banks per Device x Devices per Rank x Num of Ranks, but the characterization of N (=Banks per Device) Banks DIMM is used. For example a 4 GB 16 Banks x4 DRAM, physically, may not consists of 16 Banks only, but it is used to be refereed as a 16 Banks DIMM from the aspect of its structure characterization.

To summarize, an example table is presented with all DRAM structural characteristics expanded:

| | Example A | Example B |
|---|---|---|
| Total Size | 1 GB | 2 GB |
| Is Multi Channel | NO | YES |
| Num of DIMMS | 1 | 2 |
| DIMM Size | 1 GB | 1 GB |
| Ranks per DIMM | 1 | 1 |
| Num of Ranks | 1 | 2 |
| Devices per Rank | 16 | 8 |
| Device Size | 64 MB | 128 MB |
| Banks per Device | 16 | 16 |
| Bank Size | 4 MB | 8 MB |
| Arrays per Bank | x4 | x8 |
| Array Size | 1 MB | 1 MB |
| Rows (in bits) | 128 | 128 |
| Columns (in bits) | 64 | 64 |

Table 2.1: Example A: 1GB Memory System composed by one 1GB-x4-16Bank DIMM, Example B: 2GB Memory System composed by two 1GB-x8-16Bank DIMMs

### 2.1.2 General Simulation Techniques: Execution or Trace Driven

Let's make an overview of the memory system simulation methods and approaches so that we determine how and why a choice has been made. The most important diversifying factor, is the generation way of the memory trace because, in order to simulate the memory system behaviour and performance, a memory trace is necessary.

#### Trace Driven Simulation Approach

A tracing collection process is necessary so that the information about "interesting instructions" of a binary executable can be written out in file, the Trace File, as each instruction is executed. The tracing process can operate both over a real-existing Instruction Set Architecture (ISA), and over

an emulated and maybe not-existing one.

The only information needing to be recorded is about the data address and the type of memory operation. Trace Files have quite a large size in general ( 0,5 GB per 1 Billion Memory Accesses) for real benchmark programs. After the trace production, Trace Files are passed as input, read by the Memory System Simulator and analyzed for performance study. It should be emphasized that the Simulator can run on any machine, even though on one with different ISA, because, the Memory System Simulator itself does not need to execute the ISA but only to analyze the execution history for a particular architecture being studied.

Since a Trace-based Memory System Simulator is ISA-independent, it can be used to evaluate also not existing Architectures. In addition, this technique is very useful when the execution of a program used for trace collection, is extremely slow or wasteful, due to the fact that the Trace File is generated once and can be reused, thus, saving both resources and time.

The pros and cons of the trace driven simulation method are presented below:

**Advantages**

1. Can run on any machine as long as trace format is known.
2. Simplicity and low modeling complexity. Only memory address and access type recorded.
3. Produced once, used many times.
4. Evaluation of not existing Architectures.

**Disadvantages**

1. A Trace File is only a capture. May not be representative of the dynamic behavior of multithreaded applications. Special custom handling and synchronization needed, such as TaskSim-NANO++ [7].
2. Simplicity requires low-level of model detail. The detail lose may refer to the time accuracy, which is important for the result reliability.

3. Two step process.
4. Large disks are needed as a result of large space need to store traces.
5. High-speed disks are needed to avoid overhead from read/write traces to disk.

#### Execution Driven Simulation Approach

The Simulation follows the actual execution of a program. This can be done either with the direct execution of the program by the Memory Simulator, or with the Memory Simulator pairing with a front-end driver, which executes the program and generates the Trace. In both ways, as the Binary is executed, the trace is generated on-the-fly, so Memory Simulation is performed exactly after the execution of an Instruction. The ISA can sometimes be emulated. A timing model is also necessary for the time accuracy of the simulation. The timing model usage adds a significant overhead. The pros and cons of execution driven simulation method are presented below:

#### Advantages

1. Parallel behaviour can be captured.
2. No need to store traces.
3. No overhead from read/write to disk.
4. One step process.

#### Disadvantages

1. Execution-driven is ISA-depended, consequently increased development time for different architectures.
2. Is slower when a complex architecture is studied. (speed of the simulator is a very important issue and a complex one).
3. Time accuracy of the timing model adds a significant overhead to simulation time.

The Execution Driven approach can be implemented either *one-by-one*, or *in-fragments* as *send-to-simulator* rate. In one-by-one Execution Driven, a single instruction is approached separately, so *each* instruction is traced, it is also simulated. The Execution Driven in fragments approaches the simulation trace neither as a monolithic whole (Trace driven), nor per single instruction separately, but somewhere in the middle. The Trace is separated, during execution, into arbitrary sized fragments by storing them into a buffer. When a buffer is full-filled, it is sent for Simulation. The size of that buffer is arbitrary and must be specified experimentally.

### 2.1.3 DRAMSim2 Memory Simulator Overview

#### General

DRAMSim2 is a very popular open-source cycle-accurate JEDEC DDRx Simulator which models memory controller, memory channels, ranks, banks and all timing constraints in a general way [2]. It is developed in object-oriented C++. It can be used either in *standalone* mode to simulate memory system traces, or as a dynamic shared library which is convenient for connecting it to CPU simulators (such as MARSSx86, gem5) or other custom front-ends, in order to develop a full-system simulator.

In the current thesis DRAMSim2 is approached as the "Simulation Pair" of the Instrumentation Trace Generator Pintool from the Andronikakis thesis [9]. DRAMSim2 has the goal "to be an accurate and publicly available DDR2/3 memory system model" and can be used both for trace-driven simulations and so as Library Interface for more custom-system approaches. DRAMSim2 has came to fill the previously semi-blank field of memory system simulators with a quite simple programming interface. Despite the fact that DRAMSim2 is focused on DDR2/3, the differences with the DDR4 technology are only on latency and power so it can be used also for latency and power aimed and less rigorous DDR4 simulation.

The basic component is the cycle accurate memory controller with the mission of translation and issuing DRAM commands to a memory-bus-attached set of DRAM devices. The DRAMSim2 functionality is enclosed by an easy-to-use interface, so it is easy, also, to connect DRAMSim2

with any kind of "front-end driver", for example with a cycle accurate CPU simulator such as MARSSx86 or just a trace reader etc.

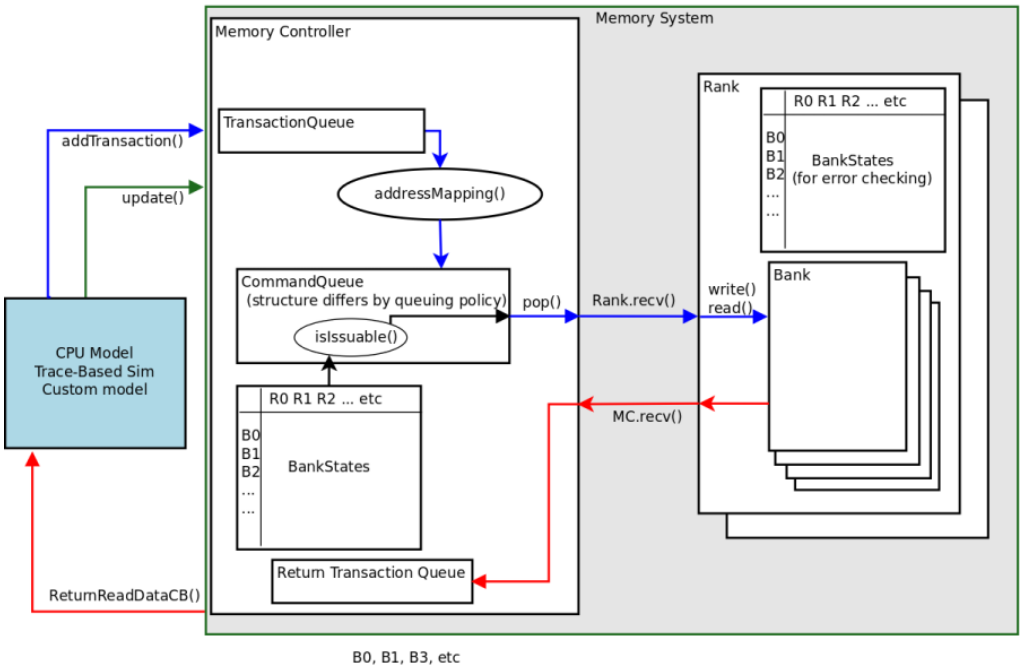


Figure 2.3: Overview of DRAMSim2

**A few words about internal functionality**

The MemorySystem class is the DRAMSim2 core-module and models one channel of a Multi-Channel Memory System. So the whole system is modelled by the MultiChannelMemorySystem class.

The constructor functions and methods of those two classes are presented so that their functionality can be explained easily :

```
1 MemorySystem::MemorySystem(unsigned id, unsigned int megsOfMemory,
      CSVWriter &csvOut_, ostream &dramsim_log_)
```

Listing 2.1: *MemorySystem* Constructor

```
1 void update();
2 bool addTransaction(Transaction *trans);
```

```
3 bool addTransaction(bool isWrite, uint64_t addr);
4 void printStats(bool finalStats);
5 bool WillAcceptTransaction();
6 void RegisterCallbacks( Callback_t *readDone, Callback_t *writeDone,
    void (*reportPower (double bgpower, double burstpower, double
    refreshpower, double actprepower) );
```

Listing 2.2: *MemorySystem* Methods

```
1 MultiChannelMemorySystem::MultiChannelMemorySystem(const string &
    deviceIniFilename_, const string &systemIniFilename_, const string &
    pwd_, const string &traceFilename_, unsigned megsOfMemory_, string *
    visFilename_, const IniReader::OverrideMap *paramOverrides)
```

Listing 2.3: *MultiChannelMemorySystem* Constructor

```
1  bool addTransaction(Transaction *trans);
2  bool addTransaction(const Transaction &trans);
3  bool addTransaction(bool isWrite, uint64_t addr);
4  bool willAcceptTransaction();
5  bool willAcceptTransaction(uint64_t addr);
6  void update();
7  void printStats(bool finalStats=false);
8  ostream &getLogFile();
9  void RegisterCallbacks(
10 TransactionCompleteCB *readDone,
11 TransactionCompleteCB *writeDone,
12 void (*reportPower)(double bgpower, double burstpower, double
     refreshpower, double actprepower));
13 int getIniBool(const std::string &field, bool *val);
14 int getIniUint(const std::string &field, unsigned int *val);
15 int getIniUint64(const std::string &field, uint64_t *val);
```

```
16 int getIniFloat(const std::string &field, float *val);
17 void InitOutputFiles(string tracefilename);
18 void setCPUClockSpeed(uint64_t cpuClkFreqHz);
```

Listing 2.4: *MultiChannelMemorySystem* Methods

The main arguments for a MultiChannelMemorySystem object are, in essence, the Device Ini file (the MultiChannelMemorySystem constructor first argument, `deviceIniFilename_`), the system ini file (the second argument, `systemIniFilename_`) and the DRAM size (fifth argument, `megsOfMemory`). With these 3 arguments we can describe our simulated Memory System structure and characteristics.

*Device Ini File* defines the *DIMM structural* characteristics for example : `DEVICE_WIDTH`, `NUM_BANKS`, `NUM_ROWS`, `NUM_COLS` and many *non-structural* characteristics such as : `REFRESH_PERIOD`, `tCK`, ,, `Vdd` etc.

*System Ini File* defines the *Memory Controller* characteristics, for example : `JEDEC_DATA_BUS_BITS`, `TRANS_QUEUE_DEPTH`, `CMD_QUEUE_DEPTH`, `SCHEDULING_POLICY`, `QUEUING_STRUCTURE` etc. It also sets the *debugging flags* of DRAMSim2.

MultiChannelMemorySystem is the DRAM system composed of one or more channels (MemorySystem). Each channel contains multiple ranks and each rank has several banks. Furthermore, each channel has a pendingTransactionQueue. In addition, there is one corresponding memory controller for each channel.

The memory controller has four queues:

1. the *Transaction Queue* (TransQ) which receives and stores incoming transactions,
2. the *Command Queue* (CmdQ) which stores the translated commands of each transaction.
3. If a read command is dispatched to the memory, then the transaction will be stored into the *Pending Read Transaction Queue* (PendRTQ) until the data is returned.
4. Finally the *Return Transaction Queue* (RtnQ) is for storing the returned transactions.

### Initialization

The first step is the creation of a new MultiChannelMemorySystem object by calling the constructor (listed in figure 2.3) with the appropriate arguments and Ini files described above. The `getMemorySystemInstance()` function can be called alternatively. That function is implemented in `MultiChannelMemorySystem` class and gets exactly the same arguments as the original Constructor function. Its task is the `MultiChannelMemorySystem` Constructor call. After that, the *Callback Functions* must be registered so as to be executed when a read or write request is completed. A simple example of initialization follows.

```
1  void main(){
2    ...
3    MultiChannelMemorySystem *mem = getMemorySystemInstance( arg1, .. );
4    TransactionCompleteCB *read_cb = new Callback< /*args*/ >(&obj, &
       some_object::read_complete);
5    TransactionCompleteCB *write_cb = new Callback< /*args*/ >(&obj, &
       some_object::write_complete);
6
7    mem->RegisterCallbacks(read_cb, write_cb);
8    ...
9    /*
10   Simulation code can follows
11   */
12   }
```

Listing 2.5: Initialization Example

### Simulation

A function must be called for every MemorySystem clock tick and also a function to queue the memory requests. After the completion of a memory request, DRAMSim2 calls the provided

callback function to inform the front-end for the completed request. The process of Simulation can be described like this: CPU requests are buffered into the TransQ in execution order, then they are translated into DRAM commands and placed into the CmdQ which can have *per rank* or *per rank per bank* structure. The memory controller takes into consideration the state of every memory bank, the read/write dependencies and the timing constraints to decide the next issue request. Due to that decision, the memory controler is, also, free to issue out-of-order. The out-of-order issuing helps to increase bank usage and bandwidth and also latency lowering. After a completed transaction, the bandwidth and latency are kept. An average calculation is performed over a given *epoch* defined in Device Ini File.

#### Refresh Modelling

Another simulation issue is the DRAM refresh. Modelling refresh is necessary because the refresh induces differentiation in the memory requests latency. If a request is issued during a DRAM refresh, it will have to wait much longer than other requests. That latency elongation can impose significant performance penalties for processors in wait for memory accesses and as a result, they significantly affect the whole system performance.

#### Power Model

Another DRAMSim2 feature is the power calculation based on the power model described in [8]. The DRAMSim2 power calculation adapts to any DDR model. The power consumption is calculated per Bank. The calculation is based in the Bank's activity where each action, or state, corresponds to a specific amount of power. That correspondence is a result of many power equations which model the DDR DRAMS and also described in [8]. This model and calculations follow in a strictly way theoretical equations and can be considered as accurate, despite any real world deviations.

### DRAMSim2 Inputs

#### Memory Trace

The Trace either as a Trace File, or as an execution driven process is the main input source of DRAMSim2. In the standalone mode the Trace is a `.trc` file, the Trace File, with 3 columns: Memory Address, Transaction Type (`P_MEM_WR`, `P_MEM_RD`, `P_FETCH`) and Cycle. In case of a front-end driver which produces Trace File timed in nano seconds, it must be preprocessed with a python parse script (comes with DRAMSim2 package) in order to be mapped to cycles.

For example:

```
0x018ADB20    P_MEM_WR    0 ps
0x018ADB28    P_MEM_WR    10.000 ns
0x01A5DB58    P_FETCH     50.000 ns
0x01A5DB50    P_MEM_RD    60.000 ns
0x01A5DB48    P_FETCH     70.000 ns
```

Listing 2.6: before parsing

```
0x018ADB20 P_MEM_WR    0
0x018ADB28 P_MEM_WR    1
0x01A5DB58 P_FETCH     5
0x01A5DB50 P_MEM_RD    6
0x01A5DB48 P_FETCH     7
```

Listing 2.7: after parsing

When used as a library interface, a simple and easy-to-use API is provided, in order to pass the transactions in a custom way . The function must be used to pass a Transaction to DRAMSim2 is the `addTransaction`, in listing 2.4.

#### System Ini File

The System Ini (system.ini) file contains the Memory System and Memory Controller parameters. They are presented below:

| | |
|---|---|
| NUM_CHANS | ADDRESS_MAPPING_SCHEME |
| NUM_RANKS | SCHEDULING_POLICY |
| JEDEC_DATA_BUS_BITS | QUEUING_STRUCTURE |
| TRANS_QUEUE_DEPTH | VIS_FILE_OUTPUT |
| CMD_QUEUE_DEPTH | USE_LOW_POWER |
| EPOCH_LENGTH | VERIFICATION_OUTPUT |
| ROW_BUFFER_POLICY | TOTAL_ROW_ACCESSES |

Some debugging flags also exist.

### Device Ini File

The Device Ini file (DRAMmodel.ini) contains all the DRAM model structural parameters such as banks, rows, columns, clock, other timing and all power parameters, as well. The most important of them are presented below:

| DRAM Device Structural parameters | |
|---|---|
| NUM_BANKS | number of banks |
| NUM_ROWS | number of array rows |
| NUM_COLS | number of array columns |
| DEVICE_WIDTH | number of arrays per device (eg x16) |
| REFRESH_PERIOD (in ns) | |
| The four Latency parameters CL-$t_{RCD}$-$t_{RP}$-$t_{RAS}$ (in cycles) | |
| CL | **CAS latency**: the number of cycles between sending a column address to the memory and the beginning of the data in response |
| $t_{RCD}$ | **Row Address to Column Address Delay**: The minimum number of clock cycles required between opening a row of memory and accessing columns within it. |
| $t_{RP}$ | **Row Precharge Time**: The minimum number of clock cycles required between issuing the precharge command and opening the next row. |
| $t_{RAS}$ | **Row Active Time**: The minimum number of clock cycles required between a row active command and issuing the precharge command. |

The parameters also include all other timing, latency and power constraints. It must be noted that other structural characteristics such as Ranks, Number of Devices per Rank etc are not defined

directly, but are calculated internally. The Device ini file defines the 4 *strategic* structural characteristics that describe directly only the DRAM Device. The Total Storage is defined as an argument in the `MultiChannelMemorySystem` Object Constructor (listing 2.3) and can be considered as the second part of the equation that calculates the remaining structural characteristics.

DRAMSim2 computes internally `PER_DEVICE_STORAGE`, `NUM_DEVICES`, and `PER_RANK_STORAGE` while all the other parameters are set by the Device Ini file.

1. `PER_DEVICE_STORAGE = NUM_ROWS * NUM_COLS * DEVICE_WIDTH * NUM_BANKS` (in bits)
2. A rank "must" have a 64 bit output bus according to the JEDEC standard, so each rank must have: `NUM_DEVICES_PER_RANK = 64 / DEVICE_WIDTH`.
   If multiple channels are ganged together, the bus width is `NUM_CHANS * 64/DEVICE_WIDTH`
3. `PER_RANK_STORAGE = PER_DEVICE_STORAGE * NUM_DEVICES_PER_RANK`
   `= NUM_ROWS * NUM_COLS * NUM_BANKS * 64`
4. MultiChannelMemorySystem object gives Total Storage in Mega Bytes so
   `NUM_RANKS = (TOTAL_STORAGE/8) / PER_RANK_STORAGE`

The only way this could run into problems is if `TOTAL_STORAGE < PER_RANK_STORAGE`. In that case DRAMSim2 sets `NUM_RANKS` = 1 and continues.

Due to that fact, it is clear that both total storage and DRAM structural parameters must be selected carefully, so that the user can be able to control the Simulation.

#### DRAMSim2 Output

Also the simulator outputs in detail, per epoch, the bandwidth, latency and power statistics to a log file. The whole output format depends on which debug flags have been set on or off.

### 2.1.4 Disaggregated Systems

The Rack-scale Disaggregated Systems is new Data Center approach, that aspires to change the traditional design, organization and building of data centers, proposing the deployment of pooled, disaggregated resources rather than "monolithic and tightly integrated components".

The Architecture of a server is traditionally organized in trays with a specific number of processors, memory size etc. Furthermore, due to their co-existence in the same node, they consist a monolithic and tightly coupled whole. The cloud or hyperscale computing servers, typically accommodate a large set of interconnected racks, each utilizing multiple interconnected server trays, as depicted in the figure above.

In general, a server tray consists of -typically multiple- CPUs attached via one or multiple Memory Controllers to tray-local Random Access Memory (RAM) for rapid instruction read and fast, random read/write byte-level access to data. The CPUs can also access persistent local storage and I/O devices (e.g. flash storage, accelerators) using a single or an hierarchy of I/O bridges.

At the same time, current Datacenter scale-out workloads mostly perform parallel tasks (e.g. internet search) that require access to vast amounts of data, but the traditional approach introduces limitations in terms of available system resources and scalability, leading to inefficiencies, suboptimal resource availability and unexploited spare resources in current datacenters.

To surmount these inefficiencies, the disaggregated architecture aims not to require memory or accelerator to be co-located with the processor in the same node. This will enhance

1. elasticity,
2. improvement of virtual machine migration, and
3. reduction of the Total Cost of Ownership (TCO)

in comparison to the current servers. In this new architecture the main building block is not the server, but the brick or dBRICK. The term brick refers to that new main building block unit in Disaggregated Architecture, which may be designed in different kinds, like compute, memory and

peripheral bricks. To build a server based on Disaggregated architecture requires, in addition, many new breakthrough developments in network, memory interface, hyper visor and orchestration layer.

For example, the network should provide ultra-low latency and high bandwidth to, efficiently, interconnect disaggregated components in the datacenter. Regarding memory, interfaces should be transparent to application. Remote-Disaggregated memory should require no changes in current applications and because of that it should be accessed as if it is local memory in today's systems.

A Disaggregated Data Center composed of computational, memory and general purpose block units.

The memory units, are the key for disaggregation. They are aimed to provide a large and flexible pool of memory resources which can be partitioned and (re)distributed among all processing nodes (and corresponding VMs) in the system. The memory units can support multiple links. These links can be used to provide more aggregate bandwidth, or can be partitioned by the orchestrator and assigned to different computational units, depending on the resource allocation policy used. This functionality can be used in two ways. First, the nodes can share the memory space of the memory unit, implementing essentially a shared memory block (albeit shared among a limited number of nodes). Second, the orchestrator can also partition the memory of the memory unit, creating private "partitions" for each client. This functionality allows for finer-grained memory allocation.

Due to the fact that DDR4 memory modules were selected for dRedBox research and development, the same memory is used in the current thesis.

## 2.2 Related Work

Memory Simulator is a topic with large body of prior work. Many approaches have been made either from the DRAM simulation aspect (Ramulator[19], DrSIM[4]), or from Memory Controller point-of-view to play a vital role on exploration of future Memory Architectures ([3]).

Ramulator as the current latest cycle-accurate DRAM simulator bears lots of improvements. Its initial idea relies on the observation that a DRAM can be abstracted as a state-machine hierarchy, with the DRAM standard dictating each state-machine behavior. It was build from scratch and

supports all the latest DRAM standards (DDR3-4,LPDDR3/4, GDDR5, WIO1/2, HBM, as well as some academic proposals (SALP, AL-DRAM, TL-DRAM, RowClone, and SARP). Finally does not sacrifice simulation speed to gain extensibility while is 2.5x faster than the next fastest simulator

The Memory Controller focused design, generally, is an alternative approach for Memory Simulation. The [3] work uses that principle to build a high-performance event-based model offering 7x simulation speed improvement without accuracy and detail level loss. That work presents a high-level memory controller model, specifically designed for full-system exploration of future system architectures. Due to the fact that is controller-focused, a DRAM memory model it is needed.

Gem5 simulator is a popular Full System Simulator containing many features, such as: Multiple CPU models, GPU model, Event-driven memory system, Multiple ISA support (Alpha, ARM, SPARC, x86) and Power and energy modeling. Regarding to the Memory Simulation using Gem5, two different memory system models are included: Classic(ast and easily configurable) and Ruby(flexible infrastructure, accurate simulation). Gem5 can play both the role of the Memory Simulator [3], or the Memory Simulator's front-end driver [2].

Due to the reason that the Intel PIN framework had already been chosen as the Simulator's front-end driver, Gem5 was not an option. Ramulator had not been released so far, so, the choice of DRAMSim2 as the memory simulation back-end was one-way. Except from that, the PIN-DRAMSim2 coupling was a contribution, since that match hadn't already been approached.

Regarding to the Last Level Cache Misses profiling, there are many already implemented tools doing so, like Cachegrind[16], OProfile[17] and perf[18]. However it was chosen the implement-from-the-scratch option using Intel Pin framework, to explore also other aspects as well.

# Chapter 3

# DIMEM Simulator: Overall Approach

## 3.1 Binary Instrumentation

In order to simulate and evaluate a memory system, except for a memory simulator, the existence of a *trace generator* or a *front-end driver* to cover the simulation traffic/input needs is also necessary. The trace generator is a software system that, as first step, produces, once, an experimental memory trace written in a Trace File, based on a real-world workload. After that, the Trace File can be used, more than once, as input in a memory simulator. The front-end driver can be a software system, also, that, for example, during a benchmark execution, takes the control of the execution and also "drives" the memory simulation. The term "drives" refers to the behavior of the front-end to correspond the memory simulation with real-time CPU-to-memory-requests following a memory trace produced on the fly (or -in other words- during execution).

For the DIMEM Simulator the pairing of a front-end driver with the DRAMSim2 memory simulator it was chosen. The development of a *Pintool* was chosen as the front-end driver. A Pintool is a program created by using Intel's PIN framework and describes how PIN will *Instrument* a specific binary executable.

The developed Pintool has the task to Instrument a binary executable to collect the memory trace of Main-Memory-only effective addresses. The memory trace is intended to *drive* the DRAMSim2

memory simulator during execution time. In a semi-parallel way, the Pintool instruments the executable until it reaches an arbitrary user-defined instructions amount, the Window, and then the collected trace is prepared to be sent for simulation to DRAMSim2. After simulation is completed, the Pintool continues that procedure iteratively until the binary execution ends.

A few more details: the Pintool gets the execution control of a binary executable program, so that it can be able to apply the *Instrumentation* and the *Analysis Functions*. The Instrumentation function is responsible to inject the instrumentation code in the binary. The Analysis function is the injected code, thus, it contains the, actually, instrumentation code.

The Pintool instruments using *per Instruction* Granularity and calls an Analysis Function when an instruction is or includes a Read or Write Operation. That is, in summary, the functionality of Instrumentation Function.

The Analysis Functions, are consequently, called when a Read or Write Operation is found. Then it is filtered through a Cache Model to decide if the Operation actually affects Main Memory. Finally, every instruction that affects Main Memory, is recorded to the Record Buffer (RB). The RB is processed in the next steps, before Simulation, for Simulation preparatory reasons. The RB reordering, approximate timing and sorting, according to the (possible) multithreading nature of the Instrumented binary executable, is the main object of the current thesis implementation.

The current thesis work has also to do with the second task of the Pintool, Simulation preparatory process and Memory System Simulation exploiting DRAMSim2. The implementation description uses code quotes to be more descriptive.

## 3.2 Memory Simulation

The implementation of simulation process of DIMEM Simulator is the main goal of the current thesis. We have already talked about the pairing of the front-end driver (Pintool) with a Memory System Simulator (DRAMSim2). In the previous section we gave a brief description of the Pintool. The object of the current section is to explain more technical details about the Memory System Simulation, the way that the capabilities of DRAMSim2 Cycle Accurate Memory System Simulator

were exploited, the implemented techniques for reordering, approximate timing and sorting, the techniques for maintenance the multithreaded nature of the trace and the simulation speed-up technique.

At the current thesis the memory system simulator was based on the "execution driven" method because of the advantages 1, 2 and 3 of the execution driven. Also, the "in pieces" simulation rate was chosen. So the rate was named Window and the whole technique "Reordering Window Technique".

### 3.2.1 The Reordering Window Technique

As Reordering Window Technique referred the way, or the rate, the collected Memory Trace is passed to the Memory Simulator. The Reordering Window Technique relies on the simulation choices have been made. This technique can be described as an hybrid of the trace and execution driven also. As first priority was the fact that Instrumentation and Simulation chosen to pair as a whole and not as a "two discrete steps" process. That choice was made because of the aim of DRAMSim2 paired with a Pintool study. So, the choice of the simple Trace File generation was rejected consequently, the trace cannot be passed to memory simulator as a Trace File, and a Buffer must be used. As mentioned above, that buffer is called RB. The RB keeps instances of Main Memory Access Record structure (MMAR), a structure to store a single instrumented instruction that affects Main Memory. An MMAR stored to RB can be passed to simulator at any time. Our choice was to be passed in fragments after a constant size of instrumented instructions.

These constant size fragments named *Windows*. The Window Size is given as an input to the Pintool and is totally experimental.

After that, because of execution-driven gained ground, it had to be a choice about the timing model. Another priority was the speed of the system in a good trade-off with the accuracy. In order to decide about the timing model with those two factors (speed and accuracy) it was proceed the choice of implementation of a simple and quite approximately timing technique, based on the additive cache latency and penalties. With that choice it is avoided the large timing model overhead. Despite that, an approximate timing exists -for sure better than no timing-. The timing

was also necessary due to another priority: the goal of multithreaded applications simulation. If the technique is not able to perform an, at least, approximate timing, it is also unable to simulate the multithreaded behavior and affection to the memory system. To summarize, the technique is an hybrid because in the side of trace generation it is not produce a Trace File, but a more dynamically generated Trace in a Buffer through a one step process, so is much closer to execution-driven. In the other side, of memory simulation, the produced trace is not using a detailed cycle accurate timing but an approximately one, so, in spite of the trace generation, from the timing perspective is much closer to trace-driven. With these characteristics, at the end of the day, we know that the reordering window technique is not the "best of all", because does not provide a true intreleaving of requests, but is the one that suits to our goals and needs: fast one-step simulation with multithreaded (at least approximate) timing capability.

### 3.2.2 Preparation of Simulation

As noted in a previous section, *Window Size* (`INS_LIMIT`) is a critical parameter of Simulation and represents the number of instructions must *pinned* before a Memory Simulation. It is noted that, the Pintool instruments until the count of *pinned* instructions equals to Window Size. It is important how the already pinned instructions are counted so that, the Pintool know when to pause instrumentation, and call Simulation functions. To count the number of Instrumented instructions it is used an `ICOUNT` class object of the `INSTLIB` namespace. The `ICOUNT` class is already implemented in Intel Pin's `icount.H` header file. The `ICOUNT` class is an independent instrumentation tool for counting Instrumented instructions and can be used with other Pintools. Because `ICOUNT` calculates the count for one thread, it was implemented a new method of `ICOUNT` class, named *MultithreadCount()*, which returns the total number of instructions already executed by every running thread. The code follows below:

```
1 UINT64 MultithreadCount() const
2 {
3         UINT64 multithreadCount = 0;
4         ASSERTX(Mode() == ModeBoth);
5         for(UINT64 i=0; i<ISIMPOINT_MAX_THREADS; i++)
```

```
6           {
7                   multithreadCount = multithreadCount + _stats[i].count;
8           }
9           return multithreadCount;
10  }
```

Listing 3.1: *MultithreadCount* Method

The implementation of that method was based on the already implemented method *Count()* which calculated the count for the caller thread only. Also it should be noted that it was implemented and another method, named *SetMultithreadCount()* for the purpose of initialization to 0 the Count. In that case, as previously, the implementation was based on the already implemented method *SetCount()* which was also designed for single thread applications. The code follows below:

```
1   VOID SetMultithreadCount(UINT64 count)
2   {
3           ASSERTX(_mode != ModeInactive);
4           for(UINT64 i=0; i<ISIMPOINT_MAX_THREADS; i++)
5           {
6                   _stats[i].count = count;
7                   _stats[i].repDuplicateCount = 0;
8           }
9   }
```

Listing 3.2: *SetMultithreadCount* Method

As soon as the number of *pinned* instructions reaches the defined Window Size (`INS_LIMIT`), the Simulation process begins.

```
1   //window size is arbitrary defined
2   #define INS_LIMIT 50000000
3   ..
4   ..
```

```
5  LOCALFUN UINT32 PinAnalysisFunction()
6  {
7          //get the instruction count so far
8          UINT32 c = icount.MultithreadCount();
9          //some analysis code
10         ...
11         ...
12         //if window size is reached go to simulation
13         if(c >= INS_LIMIT)
14         {
15                 ins_count = ins_count + c;
16                 GoToSim(threadid, c);
17                 icount.SetMultithreadCount(0);
18                 ram_count = 0;
19         }
20 }
```

Listing 3.3: Analysis Function begins Simulation

The function called by the Pin Analysis function to perform the Simulation is, at line 16, the *GoToSim()* function. That function is responsible for two discrete jobs which are implemented in the *prepareSimulation()* and the *Disaggregate()* functions. It was considered useful about these two discrete jobs to implemented as two discrete functions also, for structure programming reasons. It was estimated, likewise, that the task of calling the two functions had to be assigned in an also new/discrete function. So the GoToSim() function has a unifying role of the two discrete jobs that compose the Simulation process.

```
1  VOID GoToSim(UINT32 threadid, std::size_t arg_ins_count)
2  {
3          //print some help messages
```

```
4           ..
5           prepareSimulation();
6           Disaggregate();
7           //some more helping code
8           ...
9           return ;
10  }
```

Listing 3.4: GoToSim code

**The prepareSimulation() function**

The role of that function is to perform the necessary transformation to the RB (Trace Buffer) so that to be in the appropriate form for the simulation. The RB is a *multi*-dimensional Vector, one dimension per CPU core, and contains MMARs (Records of Instructions that affect Main Memory). The *prepareSimulation()* function performs a loop to access the RB by CPU core order. For each MMAR, calculates the *Issue-to-Memory Cycle* and then stores the Record to a new (one dimensional) Buffer named Trace Buffer (TB). The Issue-to-Memory Cycle of an Instruction (that term refers to the cycle that this instruction is going reach the Memory System and create a request) is the Issue-to-Memory Cycle of the previous Instruction from the *same* CPU core plus the current Instruction Penalty (latency).

```
rec[is].cycle = rec[is-1].cycle + rec[is].penalty;
```

After all MMARs are stored to TB, their order is not chronologically correct, so the next step is sorting by the calculated Cycle. The Sorting result is the actual Memory Trace, stored in TB, with approximate chronological order and timing, so it is ready to be Simulated. The chronological order is the sorting by Cycle result, so due to the fact that the Cycle was calculated approximately, the whole chronological order is also approximately. As a resume of the *prepareSimulation()* function we can say that receives the Instrumentation output, stored in RB, and transforms it to an approximately chronological ordered and timed Memory Trace, stored in TB, ready to be Simulated.

```
1  LOCALFUN VOID prepareSimulation()
2  {
3      PIN_MutexLock(&Mutex);
4      for (std::size_t cores=0;cores<coreNUM;cores++)
5      {
6          for (std::size_t i=0;i<j[cores];is++, i++)
7          {
8              /* store to TB(rec) and calculate cycle - recs=RB */
9              rec.push_back(Record());
10             rec[is] = recs[cores][i];
11             rec[is].threadid = recs[cores][i].threadid;
12             rec[is].ip = recs[cores][i].ip;
13             rec[is].r = recs[cores][i].r;
14             rec[is].penalty = recs[cores][i].penalty;
15             if (i==0)
16                 rec[is].cycle = 0;
17             else
18                 rec[is].cycle = rec[is-1].cycle + rec[is-1].penalty;
19             rec[is].cacheFlag = recs[cores][i].cacheFlag;
20         }
21     }
22     /* sorting by cycle */
23     std::sort(rec.begin(), rec.end(), less_than_cycle());
24     PIN_MutexUnlock(&Mutex);
25 }
```

Listing 3.5: prepareSimulation() code

The Penalty value is the additive penalty/latency of the cache hits between the current and the previous Recorded Instruction of the same CPU core.

#### The Disaggregate() function

At this point the Memory Trace is ready to be simulated. For every Instruction, firstly the Disaggregate() function is called, that decides which Memory System Model will simulate the instruction, the Local or the Remote-Disaggregated. Subsequently calls the Simulate() function with the Instruction as an attribute. In other words, the Disaggregate() function splits the Instructions through a *percentage* condition.

```
1  LOCALFUN VOID Disaggregate()
2  {
3      PIN_MutexLock(&Mutex);
4      for (std::size_t i=0;i<rec.size();i++)
5      {
6          if(i % 101 < DISAGGREGATE_PERCENTAGE)
7              Simulate(dis_mem, i, DISAGGREGATED_LATENCY );
8          else
9              Simulate(mem, i, 0);
10     }
11     PIN_MutexUnlock(&Mutex);
12 }
```

Listing 3.6: Disaggregate() code

### 3.2.3 Memory Simulation: Calling DRAMSim2

As already noted, in the current thesis, DRAMSim2 is used as an internal library interface. At the main() function of the Pintool, DRAMSim2 is initialized. The initialization registers the Callbacks of the Read, Write and everything else functions needed to be executed when a Transaction completes. These Callbacks are registered to one (or more) simulated Memory System objects, modelled by the MultiChannelMemorySystem class. In this implementation are used two discrete simulated Memory System objects, in respect with the Simulation and Evaluation of Disaggregated

Memory System approach. DRAMSim2 implementation gives the ability of simulation more than one independent Memory Systems, which may also modelled by different Device ini file. For example the first may use a x16 DDR2 and the second a x8 DDR3 etc. Each simulated Memory System produces its own output log file also. The approach of the current thesis focus more on DDR4 DRAM models, because the whole vision of Disaggregated Memory Systems will be based on the nowadays, or even future, Memory System technologies. This topic will be discussed in detail in the Evaluation and Experimental Results Chapters.

With regard to the implementation, are used two discrete Memory Systems. The first named `DIMEMmem`, and models the Local Main Memory Module. The other named `DIMEMdis_mem`, and models the Remote (or Disaggregated) Main Memory Module. When DRAMSim2 is used as a library interface, the front end driver is responsible for the time-correct passing of the input for simulation. For example, if the time between two instructions is 10 clock cycles, after the passing of the first, the front-end driver must not give the next instruction until 10 clock cycles of the memory system pass. For that purpose is the *Cycle* field, that was kept in the Instruction Record and calculated in prepareSimulation(). Cycle embodies the cache latency of the refereed instruction, and also the additive cache latency of the discarded instructions that not affect Main Memory. Using that calculation we are able to support that an approximate instruction timing is actually used, which drives the DRAMSim2 traffic.

For example we suppose the following instruction sequence:

| Core 0 | | Core 1 | |
|---|---|---|---|
| Instr. | Pen. | Instr. | Pen. |
| Hit | 1 | Miss10 | 4 |
| Miss00 | 4 | Hit | 2 |
| Hit | 1 | Miss11 | 4 |
| Miss01 | 4 | Hit | 1 |
| Hit | 2 | Hit | 3 |
| Hit | 3 | Miss12 | 4 |
| Miss02 | 4 | Miss13 | 4 |

After the front-end driver discards the Cache Hit Instructions, in prepareSimulation() is calculated the Cycle that the misses *reach* the Memory System. As explained before, the Cycle of an instruction equals with the previous instruction Cycle plus the current instruction penalty, which embodies also the additive penalty of the discarded cache hit instructions. So the table transformed:

| Core 0 | | Core 1 | |
|---|---|---|---|
| Instr. | Cycle | Instr. | Cycle |
| Miss00 | 5 | Miss10 | 4 |
| Miss01 | 10 | Miss11 | 10 |
| Miss02 | 20 | Miss12 | 18 |
| | | Miss13 | 22 |

So finally the Trace that destined for DRAMSim2 input, after chronological reordering is:

| Instr. | Cycle |
|---|---|
| Miss10 | 4 |
| Miss00 | 5 |
| Miss01 | 10 |
| Miss11 | 10 |
| Miss12 | 18 |
| Miss02 | 20 |
| Miss13 | 22 |

At this point is clear that the goal of parallel application nature maintenance in the produced Trace is achieved due to the fact that a *core regardless* reordering occurred (only by Cycle) and also may exist more than one instructions in the same cycle, which shows the parallel nature of the produced Trace.

Subsequently, the simulate() function is called. Either in the case of Local Memory, or in the case of Remote, it has the same functionality. Before the instruction is passed as a new Transaction, all the necessary Memory Updates are performed, for timing reasons. While a Memory Update, the clock, as well as the internal Memory tasks go on for a Cycle without new input. The Memory

is Updated as many times as the difference of the current and the previous instruction cycle, plus the defined Disaggregated latency. That extra latency, in the case of the local memory is equal to zero. After the updates are completed, a new Transaction is passed to Memory Simulator with its memory address and a *is read or write* boolean value as arguments. The code follows below:

```
1  /*      mem_ is the simulated Memory System (local or remote)
2          i is the Trace Buffer index
3          dis_latency is the extra latency for remote
4  */
5  LOCALFUN VOID Simulate( MultiChannelMemorySystem *mem_, std::size_t i,
       std::size_t dis_latency)
6  {
7      bool isWrite;
8      /* to avoid out-of-border access the first time */
9      if (i == 0)
10     {
11         for (std::size_t i2=0; i2 < rec[i].cycle - 0 + dis_latency; i2
               ++)
12             mem_ -> update();
13     }
14     else
15     {
16         for (std::size_t i2=0; i2 < rec[i].cycle - rec[i-1].cycle +
               dis_latency; i2++)
17             mem_ -> update();
18     }
19     /* is read or write find out */
20     isWrite = (rec[i].r == 'W' ? true : false);
21     /* add new Transaction to Simulator */
22     obj.add_one_and_run(mem_, rec[i].ip, isWrite);
```

```
23 }
```

Listing 3.7: Simulate() code

The `add_one_and_run()` is the function that, finally, begins the simulation by passing a new Transaction to the simulated Memory System. It calls the `addTransaction()` function of DRAMSim2 with `isWrite` and `addr` as attributes. Essentially, that `addTransaction()` refers to the MultiChannelMemorySystem object method `addTransaction()`, which calls the corresponding MemorySystem `addTransaction()`. If, at this point, the Transaction becomes acceptable, directly is added to MemoryController `Transaction Queue` and then translated to DRAM command so as to be added to `Command Queue` and executed when its order comes; if not, is added to MemorySystem `Pending Queue` until is acceptable.

Those actions followed iteratively as soon as the whole Trace Window is passed for simulation. When the simulation session completes, the Pintool turns back to *Instrumentation mode* for the next Window etc. The whole process completes when the binary executable terminates, which means that the whole set of Instructions affect Main Memory is Instrumented and Simulated.

## 3.3 Skip Mode Feature

The applications run for Billion of Instruction and Cycles, and they have significant initialization phases. We therefore need the ability to "skip" initialization and simulate the "core" of the application or to sample over the application execution.

The choice of Skip mode can be applied to many tasks of the DIMEM Simulator, but it was observed that just as the Instrumentation sub-task has large overhead, so as the Simulation sub-task has the same, so it was decided the Skip mode to be applied over a full (Instrumented + Simulated) processed Window *periodically.*

The additional implementation based on a user defined constant integer named `SKIP_MODE` , which models the number of periodically skipped windows. For example if `SKIP_MODE` is 5, the DIMEM

Simulator will be in Skip mode for 5 Windows, before simulates one. Subsequently, again skips 5 Windows, one simulated etc.

New code added at Instrumentation Analysis functions, so as to be checked if the DIMEM Simulator is in Skip or Normal Mode before the sub-tasks, which may must be skipped, actually executed or called. It should be added into these functions because they are responsible for either Instrumentation or Simulation sub-task calls.

```
1  LOCALFUN UINT32 AnInstrumentationAnalysisFunc(arg1, arg2, ...)
2  {
3      /* calculate window # */
4      UINT32 c = icount.MultithreadCount();
5      UINT32 windowCnt = ins_count / INS_LIMIT;
6
7      if( ( windowCnt >= STARTING_WINDOW-1 ) && \
8            ( (windowCnt % (SKIP_STEP + 1)) == 0 || SKIP_STEP == 0 ) && \
9                (simulated_windows_counter < WINDOWS_TO_SIMULATE) ) \
10     {
11         /* instrumentation analysis code */
12         ...
13     }
14     else
15     {
16         /* Skip mode code
17            print some helping messages */
18         ...
19         icount.SetMultithreadCount(0);
20     }
21 }
```

Listing 3.8: Skip Mode code

## 3.4 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: 132980
      Ram Percentage: 0.27%
      Simulation number: 1 of 40
=====================================

      ...
      Local Clock: 584842980
      Disaggregated Clock: 6708072956

      Total Instructions: 6300001184
      Total Instrumented Instructions: 2000000000

      0.292 cyles per Instruction (L)
      3.35 cyles per Instruction (D)
      ...
      L1 Data Cache 0:
              Load Hits:        126941409
              Load Misses:        7090283
              Load Accesses:    134031692
```

```
Load Miss Rate:       5.29%

Store Hits:          61278276
Store Misses:         4117665
Store Accesses:      65395941
Store Miss Rate:      6.30%

Total Hits:         188219685
Total Misses:        11207948
Total Accesses:     199427633
Total Miss Rate:      5.62%
Flushes:                    0
Stat Resets:                0
```

Listing 3.9: 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: BARNES | CPU: IVY | Scenario: 25-75 | Step: 3 |
 ================================================================

 ============== Printing Statistics [id:0]==================
Total Return Transactions : 5521948 average bandwidth 68.119MB/s
 -Rank 0 :
       -Reads  : 462380 (29592320 bytes)
   -Writes : 184524 (11809536 bytes)
```

```
   -Bandwidth / Latency  (Bank 0): 4.282 MB/s  33.576 ms
   -Bandwidth / Latency  (Bank 1): 4.300 MB/s  33.401 ms
   -Bandwidth / Latency  (Bank 2): 4.283 MB/s  33.555 ms
   -Bandwidth / Latency  (Bank 3): 4.249 MB/s  33.441 ms
   -Bandwidth / Latency  (Bank 4): 4.227 MB/s  33.471 ms
== Power Data for Rank        0
  Average Power (mW)      : 1000.132
    -Background (mW)      : 976.925
    -Act/Pre    (mW)      : 7.577
    -Burst      (mW)      : 13.621
    -Refresh    (mW)      : 2.009
-Rank 1 :
...
---  Latency list (17)
      [lat] : #
      [30-39] : 5509845
      [40-49] : 6095
      [50-59] : 6008
== Pending Transactions : 0 (9081920616)==
```

Listing 3.10: DRAMSim2.out example

The number in brackets at the last line represents the reached DRAM Clock Cycle.

# Chapter 4

# Evaluation and Experimental Results

In that chapter the CPU, DRAM and Benchmark choices are reported. The ISA choices are described within the CPU metrics, the used DRAM model is described in the DRAM Metrics and a summary of Splash-3 and PARSEC benchmark suites is presented.

## 4.1 CPU Configurations and Metrics

As it has already been explained, the DIMEM Memory 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 modern and popular CPUs for HPC, are used in the current thesis:

1. Intel Ivy Bridge i7 3770
2. Intel Skylake i7 6700K
3. Intel Xeon X5-650

Their metrics are presented below, as well as their corresponding cache metrics:

| | Ivy Bridge i7 3770 | Skylake i7 6700K | Xeon X5-650 |
|---|---|---|---|
| Cores | 4 | 4 | 6 |
| Hyperthreading | 2 threads/core | 2 threads/core | 2 threads/core |
| Logical Cores | 4 | 4 | 6 |
| Cache | 3-levels and TLB | 3-levels and TLB | 3-levels and TLB |

Table 4.1: The general specs table of the 3 CPUs

| Intel Ivy Bridge i7 3770 | | | | | |
|---|---|---|---|---|---|
| TLB | | Cache Levels | | | |
| | DTLB | Levels | L1 | L2 | UL3 |
| Items | 32 | Size | 32 KB | 256 KB | 8 MB |
| Line Size | 2 MB | Line Size | 64 B | 64 B | 64 B |
| Associativity | 4-way | Associativity | 8-way | 8-way | 16-way |
| Penalty | 16 | Penalty | 5 | 12 | 30 |

Table 4.2: Intel Ivy Bridge i7 3770 Cache Metrics

| Intel Skylake i7 6700K | | | | | |
|---|---|---|---|---|---|
| TLB | | Cache Levels | | | |
| | DTLB | Levels | L1 | L2 | UL3 |
| Items | 64 | Size | 32 KB | 256 KB | 8 MB |
| Line Size | 4 KB | Line Size | 64 B | 64 B | 64 B |
| Associativity | 4-way | Associativity | 8-way | 4-way | 16-way |
| Penalty | 9 | Penalty | 4 | 12 | 42 |

Table 4.3: Intel Skylake i7 6700K Cache Metrics

| Intel Xeon X5-650 | | | | | |
|---|---|---|---|---|---|
| TLB | | Cache Levels | | | |
| | DTLB | Levels | L1 | L2 | UL3 |
| Items | 64 MB | Size | 32 KB | 256 KB | 16 MB |
| Line Size | 2 MB | Line Size | 64 B | 64 B | 64 B |
| Associativity | 4-way | Associativity | 8-way | 8-way | 16-way |
| Penalty | 16 | Penalty | 4 | 10 | 41 |

Table 4.4: Intel Xeon Cache Metrics

## 4.2 DRAM Configurations and Metrics

The Memory System is portable, as well as, the CPU. The user can modify and experiment with his/her own choices. As described in DRAMSim2 overview section in Chapter 2 the Device Ini and System Ini contain many variables which can be modified. A modified Micron MT40A1G4HX-083E DDR4 SDRAM[15] 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 4.5: DRAM Metrics - Modified DDR4 Micron 1G 16B x4 sg083E

## 4.3 Splash-3 Benchmark Suite

The Stanford ParalleL Applications for SHared Memory (SPLASH) is a well-known parallel application suite which had been widely used as workload in many architectural studies.

In 1995 the suite was expanded to include several new programs as well as original improved versions. The resulted new version was SPLASH-2 [12] which has also been widely used in research.

However, Splash-2 was released over 20 years ago and does not follow the recent C memory consistency model. The modern and nowadays used compilers and hardware, lead Splash-2 benchmarks to unexpected behavior and often, also, incorrect output. The Splash-3 Benchmark suite [13] was released on that basis. That "updated" Splash suite has rectified the problematic benchmarks and contributes to the community a new sanitized version of the Splash-2 benchmarks.

The main Splash-3 improvement is the optimization of the data races. The Splash-2 data races are not perceived as a bug, but as an effort to enhance scalability for the older machine generations where synchronization, such as locks, mutexes and conditional variables, were not considered as inexpensive as today.

As described in [13] it was observed that data races "can lead to unexpected and often incorrect behavior when the applications containing them are used in conjunction with contemporary C compiler or hardware (either simulated or real) that supports a more relaxed memory model than TSO, as for example Release Consistency (RC) [13] or a Weak memory model (e.g., ARM [14] or Power [15] architectures). Unexpected behaviors are translated to non-deterministic or incorrect output, or even to performance bugs". As a result, additional synchronization, either with locks or with conditional variables (signal/broadcast/wait), was added where was necessary.

For the evaluation, finally, three benchmarks from the Splash-3 suite were used : **Barnes**, **Volrend** and **Raytrace**.

#### Input Sets

The input sets which were used, are offered by PARSEC benchmark suite, which also includes a Splash distribution. That benchmark suite defines and refers to a **size-classified** input set system, composed by six discrete input set sizes. The **test** size, for program's functionality testing, the **simdev** size, that has a behavior similar to the real one (also for test and development), the **simsmall**, **simmedium**, **simlarge** sizes which are for simulators and microarchitectural studies, and finally the **native** size which is designed, and literally used, for real machine performance measurements.

From our aspect, the **native** input set is the most interesting one due to the reason that is the most similar to real program inputs.

### Barnes

The Barnes application performs a simulation of a N-body system (galaxies or particles, for example) interaction, using Barnes-Hut hierarchical N-body method. The simulation is carried out in three dimensions over a number of time-steps. For the evaluation process it was used with the native workload, which is presented below:

| Barnes Parameters | Native Input Values |
|---|---|
| Number of particles/bodies (int) | 4194304 |
| The seed used by the random number generator (int) | 123 |
| The time-step (double) | 0.025 |
| The usual potential softening (double) | 0.05 |
| The cell subdivision tolerance (double) | 1.0 |
| Cells per Fleave (double) | 2.0 |
| Number of Fleaves (double) | 5.0 |
| The time to stop integration (double) | 0.075 |
| The data-output interval (double) | 0.25 |
| The number of processors (int) | 64 |

Table 4.6: Barnes Native Input

### Volrend

This application a three-dimensional volume rendering making use of a ray casting technique. A cube of voxels (volume elements) represents the volume, which is traversed in a quickly way by an octree data structure. The program renders several frames from changing viewpoints. The benchmark gets the number of Threads and the model file, as inputs. But it can be configurable by changing its parameters in the code. In order to set up a native, the parameters above are configured:

### Raytrace

This application uses ray tracing technique to render a 3-D scene. The scene is represented by an hierarchical uniform grid (similar to an octree). It also implements antialiasing early ray termination. A ray is traced through each pixel in the image plane, and reflects in upredictable ways off the objects it strikes.

| Volrend Native Input | |
|---|---|
| Runtime parameters | |
| Thread Number | 64 |
| Input Model | head.den |
| user-options.h file parameters | |
| BLOCK LEN (image block size) | 1024 |
| const.h file parameters | |
| ROTATE STEPS | 10000 |
| STEP SIZE | 3 |

Table 4.7: Volrend Native Input

| Raytrace Parameters | Native Input Values |
|---|---|
| Number of Threads | 64 |
| Antialiasing | 128 |
| Shared Memory | 96 MB |
| Object Model | Car |

Table 4.8: Raytrace Native Input

## 4.4 PARSEC Benchmark Suite

A cooperation between Intel and Princeton University aiming to a Benchmark Suite for Chip Multiprocessors, was resulted at **P**rinceton **A**pplication **R**epository for **S**hared-M**e**mory **C**omputers or PARSEC Benchmark Suite[14]. It is an open source parallel benchmark suite of emerging applications for evaluating multicore and multiprocessor systems. It contains a wide range of application domains: financial, computer vision, physical modeling, future media, content-based search and deduplication. Its current release is 3.0 (since summer 2011).

For the evaluation, finally, it was used the Fluidanimate benchmark.

#### Fluidanimate

It is an Intel's Computer Animation application, which simulates the underlying physics of fluid motion (smoke and particles effects) for realtime animation purposes with Smoothed-Particle Hydrodynamics (SPH) Algorithm. Physics simulations allows significantly more realistic animations, so that it is a highly demanded feature for games. Fluid animation, more specificaly, is one of the most challenging effects and it starting to get used in games.

| FluidAnimate Parameters | Native Input Values |
|---|---|
| Number of Threads | 64 |
| Number of Frames | 100 |
| Input File | in500K.fluid |

Table 4.9: FluidAnimate Native Input

The input file is the native input set, as characterized by Parsec 2.1 package and is a list of 500K particles.

## 4.5 Experimental Process

In that section the experimental process strategy is going to be described. The result reliability achievement was the main problem of the experimental process design. As mentioned in chapter 3, the DIMEM Simulator uses Skip mode, because of the time consuming simulation as a result of the large overhead over the benchmark execution time. Skip mode, in essence, is a sampling method, consequently, the existence of the statistical error $\varepsilon$ 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.

As described earlier, the chosen simulation sample size is 2 Billion benchmark instructions. 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 usefull 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 [19] 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 $\varepsilon$.

To calculate the $\varepsilon$, a complete simulation, without sampling, is a one-way street. But as a result of the time limitations, imposed by the complete simulation, (expected duration $\sim$ 90 days = 3 months), the solution of just a 4 times denser simulation has been chosen. A simulation with higher density will not provide us with an accurate $\varepsilon$, 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. For example we assume that the benchmark profile shows a spike (many cache misses) which occupies the 15% of the benchmark behaviour and the rest 75% is a quite constant line. To simulate in a representative way, one must utilize the 15% of the samples to be spread over the spike, as well as, the rest 75% over the constant line. In order to achieve the corresponding and appropriate sampling distribution, each benchmark was utilized separately. The *dense method* was used only for Barnes benchmark. It was the only one whose behaviour was notably complicated, with hard-to-catch spikes etc. That behaviour can easily leads us to unreliable results, so an indication of the result correctness level it is very helpful. Many different behaviours have been observed between the benchmarks and as well as their utilization, are presented below:

**Barnes Profiling and Utilization**

The Barnes benchmark profile presents a periodic behavior, and each period can be divided in three discrete areas. The first is the *spike* with $\sim$45 miss per kilo instructions (MPKI) and occupies 0.4% of a period. Second, is the area before the spike with $\sim$2 MPKI and occupies the 4% of a period. Finally is the rest 95.6% area with $\ll$ 1 MPKI. The experiment utilization below:

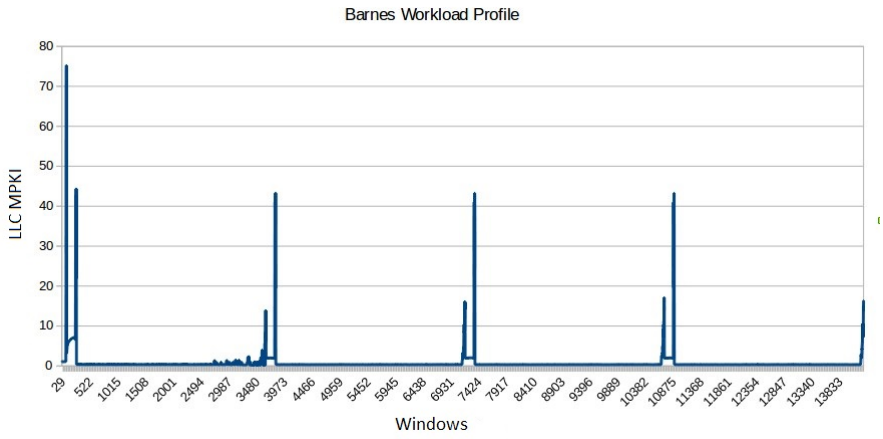


Figure 4.1: Barnes Profile

| | | Number of Samples | |
|---|---|---|---|
| Area | Percentage | 2B (Simple) | 8B (Dense) |
| Spike | 0.4% | 1 | 4 |
| Before Spike | 4% | 2 | 8 |
| Rest | 95.6% | 37 | 148 |

Table 4.10: Barnes Utilization

**Volrend Profiling and Utilization**

An alternative utilization approach has been chosen for Volrend. During profile study (figure 4.2a), it was observed that it looks like a periodic function. Due to that observation, a Fourier transformation was applied, to compute the signal over the frequency spectrum. For simplicity, one second per Window it was set as the time measurement unit.

The spectrum in 4.2b shows that the Volrend profile is not a simple periodic function but a complex one. That means that more than one periodic functions are included, subsequently the strictly mathematical period is hard to find out. The Volrend program follows an iterative process, and due to cache locality effects, a slightly different curve is formed in the last level cache misses profile diagram, despite the fact that the profile curves refer to the same iterative process.

With Nyquist sampling theorem, given the Maximum spectrum frequency, we can calculate which is the sampling frequency threshold, in order to produce reliable simulation results. The max frequency is 0.0145 Hz, subsequently, we must use a sampling frequency $\geq$ 0.029 Hz according to

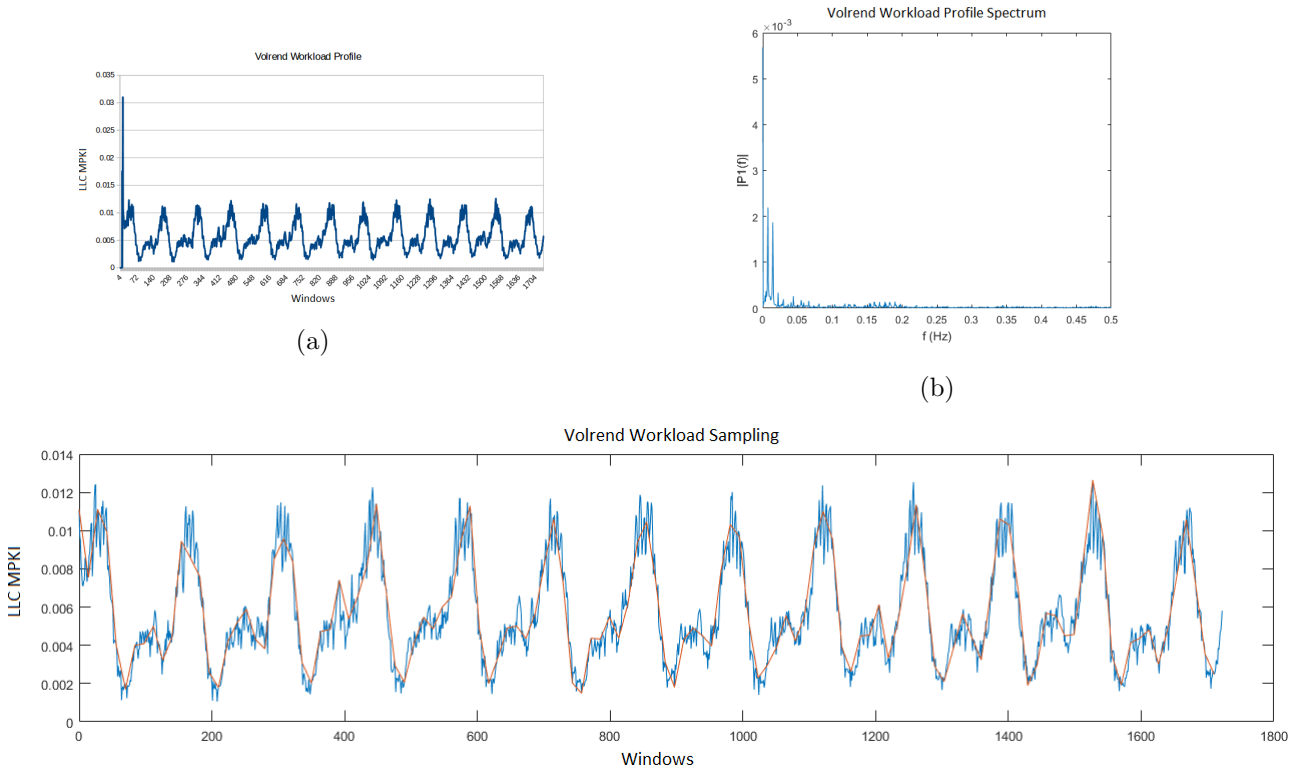


Figure 4.2: Volrend Profile Sampling

Nyquist sampling theorem.

In that benchmark we use 200 Million Instructions sized Windows, subsequently, to perform a 2 Billion Instructions Simulation, we need 10 Windows (samples). Given that amount, if we sample over a *period* (consists of ∼140 Windows), the sampling frequency is 0.0714 Hz and overpass the Nyquist threshold frequency. The sampling trace is illustrated with the orange line in figure 4.2c over the blue one (initial workload profile).

The needed utilization to operate that concept is: simulation with skip Mode 14 after the initialization phase of the Benchmark.

### Raytrace Profiling and Utilization

Raytrace Workload profile doesn't show any periodic behaviour, in contrast with Barnes and Volrend profiles. As we can see in figure 4.3, the diagram can be divided in three discrete areas. The

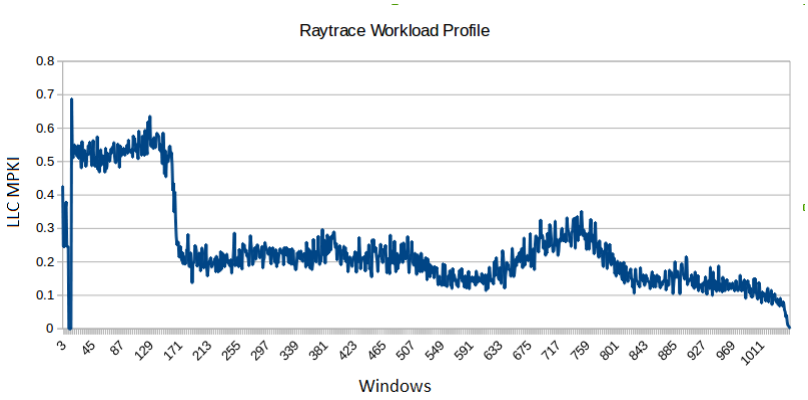


Figure 4.3: Raytrace Profile

first area, with percentage 12%, lays after initialization phase to Window 170 , the second area, with percentage 23%, is the pyramid-shaped area between Windows 600 and 850. The rest can be perceived as the third area, with percentage 65%.

Due to that behaviour (no spikes or periods) it was assumed that if the simulation sampling follows the ratio between the areas, the results will be quite reliable. For simplicity reasons no more dense experiments were made. It was also judged that, a possible denser experiment, wouldn't be able to give us a more accurate indication, about the result accuracy level, than the dense Barnes did. The experiment utilization below:

| Area | Percentage | Number of Samples |
|---|---|---|
| First | 12% | 5 |
| Pyramid | 23% | 9 |
| Rest | 65% | 26 |

Table 4.11: Raytrace Utilization

### FluidAnimate Profiling and Utilization

FluidAnimate Workload profile shows no periodic behaviour as Raytrace's profile does. As we can see in figure 4.4, the diagram cannot be divided in discrete areas. We observe that the majority of the values lying bellow 3 MPKI, and many others over 14 MPKI.

Due to profile non distinctness (no periods or discrete areas) it was prefered to simulate a solid area. In that benchmark we use 50 Million Instructions sized Windows, subsequently, to perform a

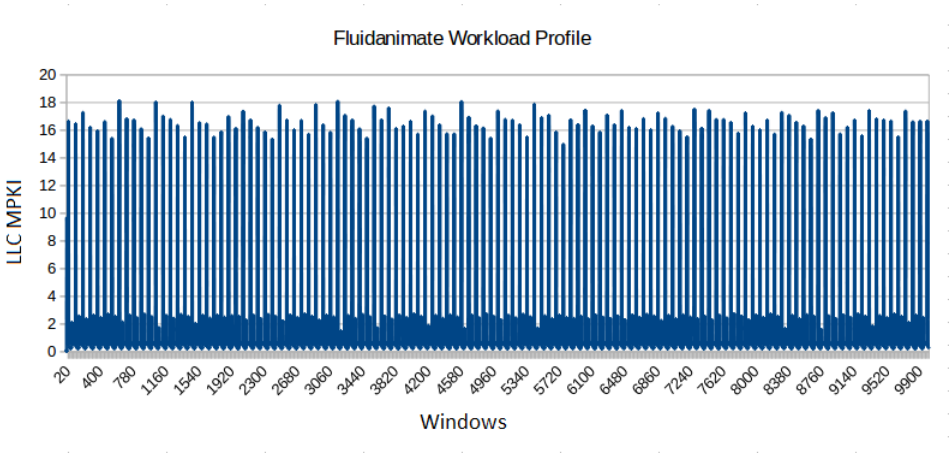


Figure 4.4: FluidAnimate Profile

2 Billion Instructions Simulation, we need 40 Windows (samples). After experimental observation, the chosen area laid after initialization phase and included both samples below 3 MPKI and spikes over 14 MPKI.

### 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 4.12: Nanosecond to Memory Cycles Disaggregated Latency Correspondence

## 4.6 Experimental Results

The experimental results are presented per Benchmark, CPU and Simulation Scenario. First are illustrated the results for Barnes benchmark with Ivy Bridge i7 3770, then with Skylake i7 6700K and finally with Xeon X5-650. Subsequently, the results for Volrend, Raytrace and FluidAnimate 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 behaviour 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 behaviour 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.

#### Dense Barnes Experimental Results

As noticed earlier, a denser experiment for Barnes has been performed, in order to evaluate that Benchmark's experimental process. The Barnes Benchmark profile shows some spikes which are hard-to-catch during simulation. Because of that, it had to be determined in a specific way the Windows where the simulation would be applied to. For 2 Billion Instructions Simulation size, the spike area corresponds to only one Window Simulation, in contrast with the denser Simulation, in which corresponds to four. The Dense Experiment results show 17% divergence in comparison with the 2 Billion Instructions Experiment. Despite the fact that the Disaggregated behaviour is also depicted as like the 2 Billion Experiment, it can be understood that for a benchmark like Barnes, which shows a spiky LLC MPKI profile, the 2 Billion Instructions Simulation are not enough. For a spiky profile, a more representative Simulation sample it is needed.

#### Results Comparison

A comparison between the Benchmarks about the Disaggregated Latency impact follows. A table per benchmark containing the 500ns (lowest) and 2000ns (highest) impact, as well as, an average one are presented:

The less Disaggregated Latency impact per Scenario belongs to Barnes Ivy for 75-25, Barnes Xeon for 50-50 and Raytrace Ivy for 25-75. The Avergate table shows that the Barnes Benchmark is influenced the less for all Scenarios. On the other hand, FluidAnimate is influenced the most for 75-25 and 25-75 Scenarios, as well as Raytrace for 50-50 Scenario. The Volrend Benchmark is out of

| Bench | Ivy | | |
|---|---|---|---|
| | 75-25 | 50-50 | 25-75 |
| Barnes | [+10, +30] | [+20, +70] | [+25, +110] |
| Volrend | [-5, -5] | [-5, -5] | [-5, -5] |
| Raytrace | [+20, +40] | [+25, +60] | [+30, +80] |
| FluidAnimate | [+10, +70] | [+25, +140] | [+40, +210] |

Table 4.13: Disaggregated Latency impact range - Ivy

| Bench | Skylake | | |
|---|---|---|---|
| | 75-25 | 50-50 | 25-75 |
| Barnes | [+20,+80] | [-10,+70] | [+50,+250] |
| Volrend | [-5, -5] | [-5, -5] | [-5, -5] |
| Raytrace | [+25, +120] | [+50, +220] | [+70, +350] |
| FluidAnimate | [+25, +120] | [+10, +125] | [+70, +350] |

Table 4.14: Disaggregated Latency impact range - Skylake

| Bench | Xeon | | |
|---|---|---|---|
| | 75-25 | 50-50 | 25-75 |
| Barnes | [+10,+60] | [-10,+60] | [+60,+220] |
| Volrend | [-5, -5] | [-5, -5] | [-5, -5] |
| Raytrace | [+10, +70] | [+200,+450] | [+50, +350] |
| FluidAnimate | [+10, +70] | [+20, +125] | [+50, +250] |

Table 4.15: Disaggregated Latency impact range - Xeon

| Bench | Average | | |
|---|---|---|---|
| | 75-25 | 50-50 | 25-75 |
| Barnes | 56% | 65% | 190% |
| Volrend | -5% | -5% | -5% |
| Raytrace | 75% | 240% | 230% |
| FluidAnimate | 85% | 140% | 270% |

Table 4.16: Average Disaggregated Latency Impact per Benchmark

that comparison due to the fact that, it presents a constant behaviour below the 100-0 (Local-only) *limit*. That may happens because, as the LLC Misses profile showed, the benchmark's workload lies too low, ranging between 0,002 and 0,012 MPKI.


Figure 4.5: Barnes Ivy Results


Figure 4.6: Barnes Skylake Results


Figure 4.7: Barnes Xeon Results

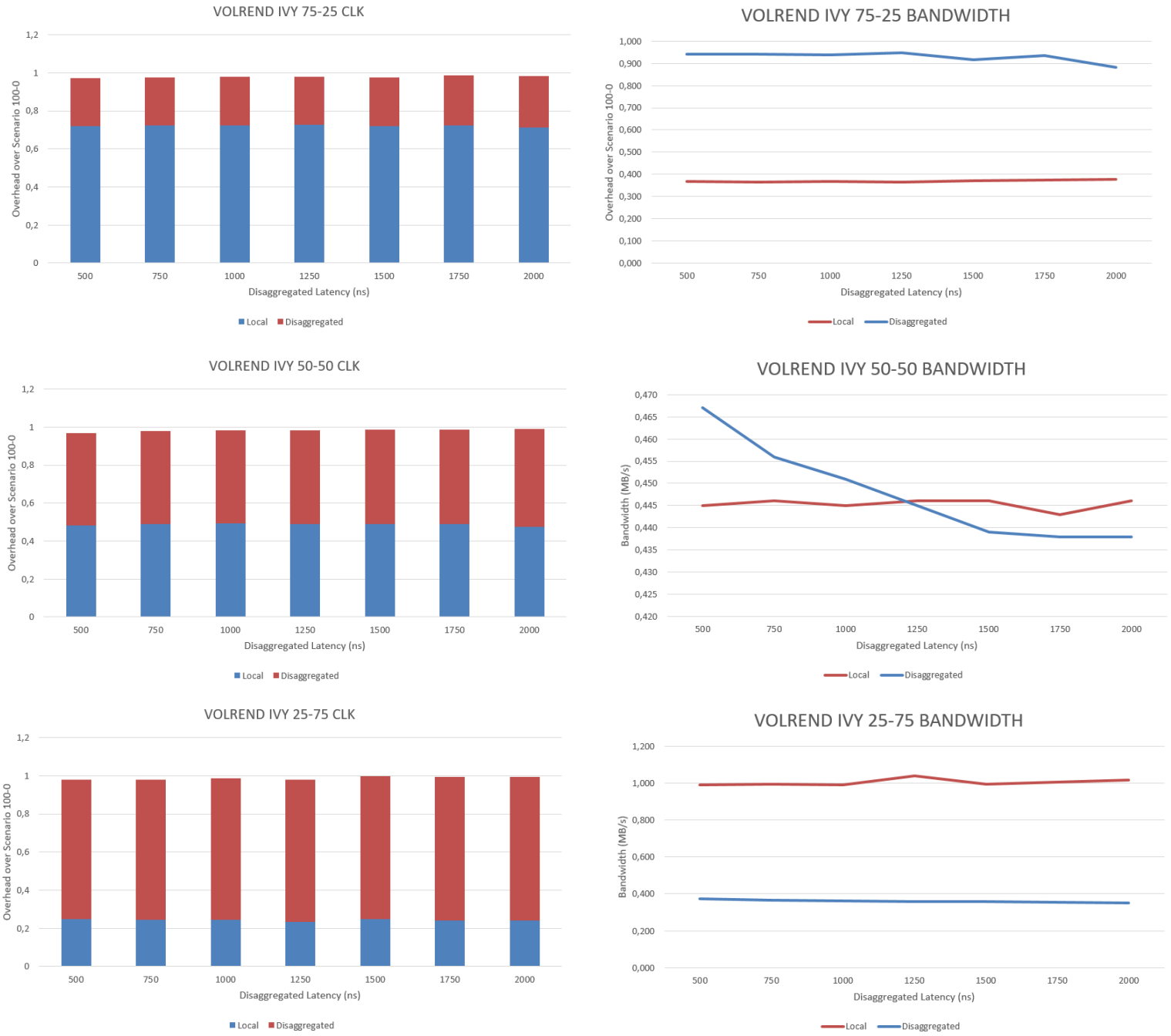


Figure 4.8: Volrend Ivy Results

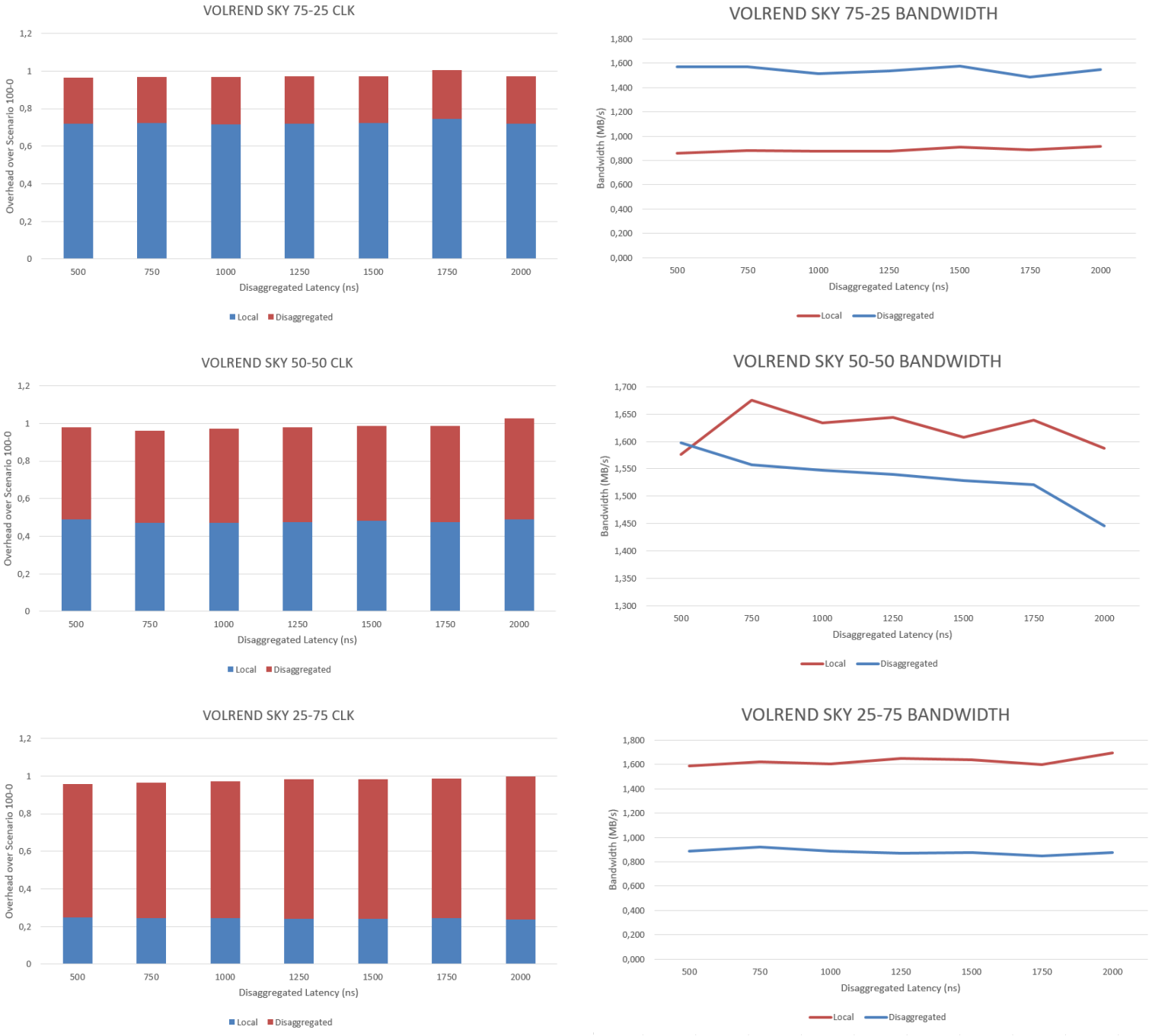


Figure 4.9: Volrend Skylake Results


Figure 4.10: Volrend Xeon Results


Figure 4.11: Raytrace Ivy Results

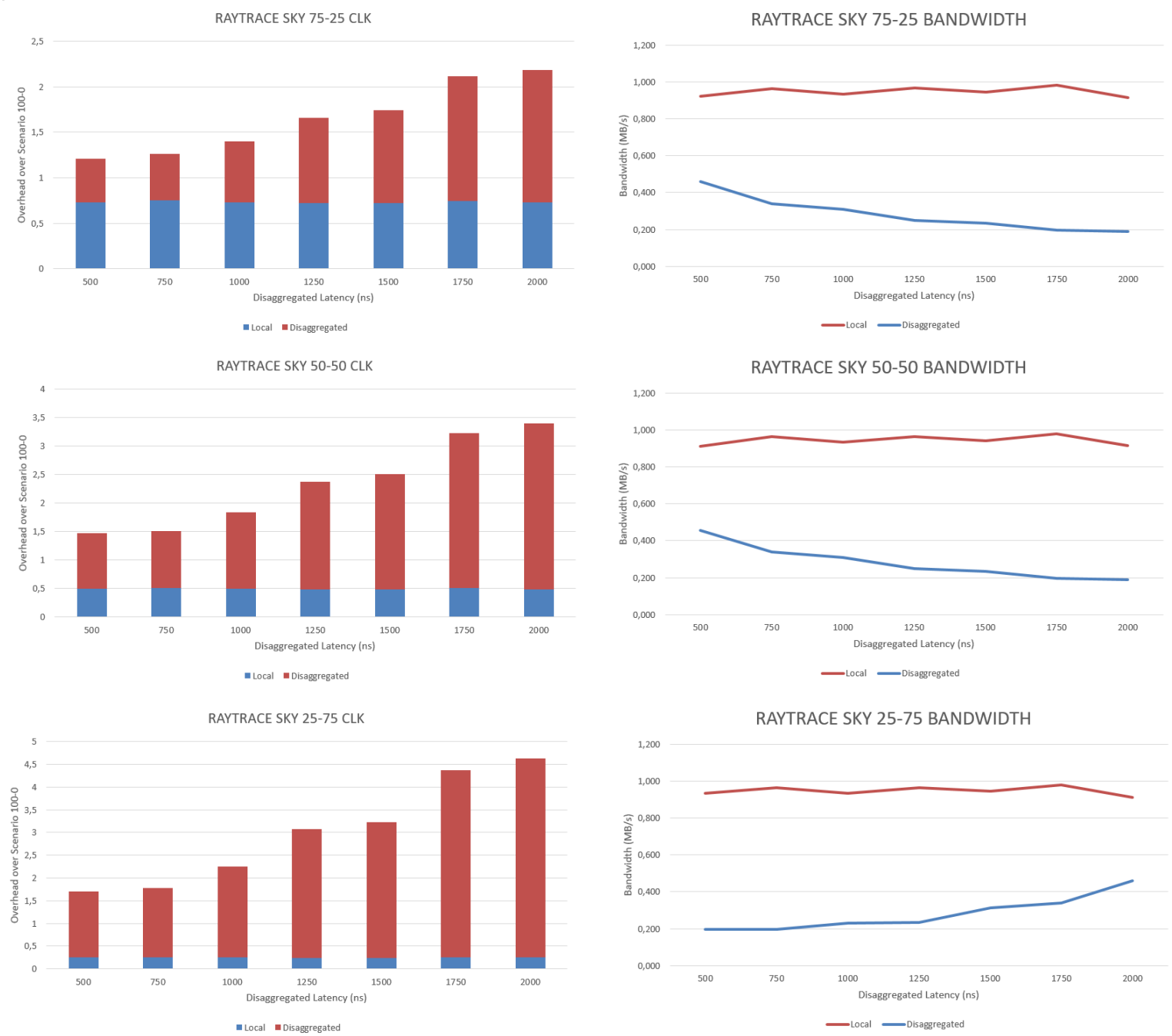


Figure 4.12: Raytrace Skylake Results


Figure 4.13: Raytrace Xeon Results


Figure 4.14: FluidAnimate Ivy Results

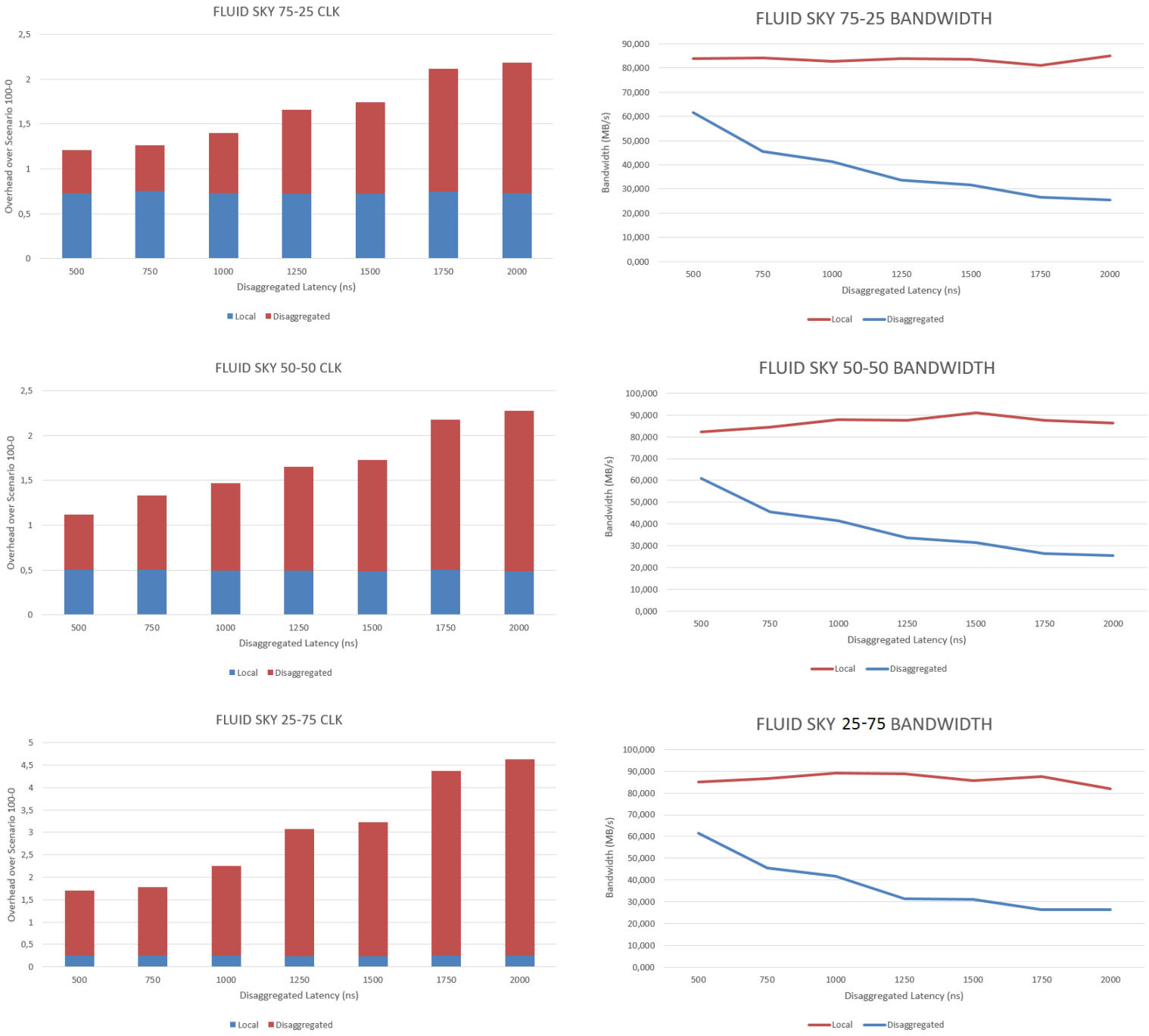


Figure 4.15: FluidAnimate Skylake Results

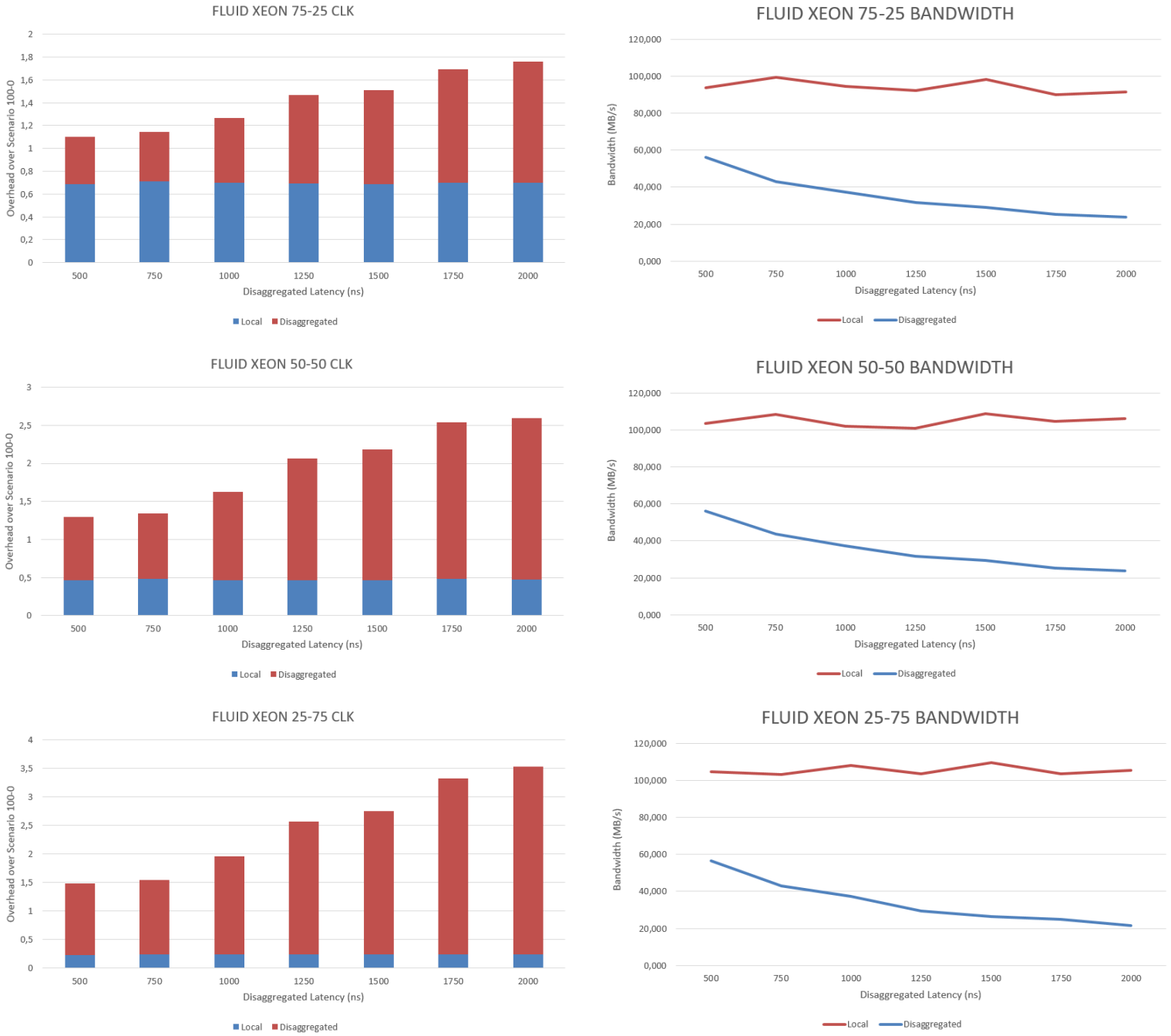


Figure 4.16: FluidAnimate Xeon Results

# Chapter 5

# Conclusion and Future Work

## 5.1 Conclusion

Concluding we can support that we have developed a simple Disaggregated Memory System Simulator. That approach comes with a level of detail, able to depicts the Disaggregated Memory System behaviour, 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 DIMEM Simulator has never developed before, not even in a simplistic way.

Regarding to the experimental methods, we can say that different approaches were used. To simulate in a sampling way over a benchmark application, the profile knowledge is very important. Sampling with Nyquist theorem seems more safe, accurate and reliable, than the more custom way. However the custom way sampling is not meaningless. In the current thesis approach we just observed that a more representative sampling amount had to be chosen.

Furthermore, we observe that the Disaggregated Memory System behaviour 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.

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.

## 5.2 Future Work

1. 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.
2. 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.
3. Further study and experimentation with more DRAMSim2 parameters.
4. Experimentation with another DRAM Simulators. Ramulator[19] 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) architecture. HMC-Sim2.0[20] and CasHMC[21] 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.

# Bibliography


[1] B.Jacob, S.W.Ng, D.T.Wang, *"Memory Systems: Cache, DRAM, Disk"*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2007.

[2] P.Rosenfeld, E.Cooper-Balis, B.Jacob, *"DRAMSim2: A Cycle Accurate Memory System Simulator"*. IEEE Computer Architecture Letters, pp. 16-19, 2011.

[3] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, A.N. Udipi, *"Simulating DRAM controllers for future system architecture exploration"*. Performance Analysis of Systems and Software (ISPASS), 2014.

[4] M.K. Jeong, D.H. Yoon, and Mattan Erez. *"DrSim: A Platform for Flexible DRAM System Research"*.
`http://lph.ece.utexas.edu/public/DrSim`

[5] N.Binkert et all, *"The gem5 Simulator"*. ACM SIGARCH Computer Architecture News, vol 39, pg: 1-7, 2011.

[6] D.T.Wang, *"Modern DRAM Memory Systems: Performance Analysis And Scheduling Algorithm"*. Doctor of Philosophy Dissertation, University of Maryland, 2005.

[7] A.Rico, A.Duran, E.Cabarcas, Y.Etsion, A.Ramirez, M.Valero, *"Trace-driven Simulation of Multithreaded Applications"*. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pg: 87-96, April 2011.

[8] Micron Technology, *"TN-46-03 Calculating Memory System Power for DDR"*. Tech. Rep., 2001.

[9] Andreas Andronikakis, *"Memory System Evaluation of Disaggregated Cloud Data Centers"*. Diploma Thesis, Technical University of Crete, Chania, May 2017.

[10] K.Katrinis, D.Syrivelis, D.Pnevmatikatos, G.Zervas, D.Theodoropoulos, I.Koutsopoulos, K.Hasharoni, D.Raho, C.Pinto, F.Espina, S.Lopez-Buedo, Q.Chen, M.Nemirovsky, D.Roca, H.Klos, T.Berends, *"Rack-scale Disaggregated cloud data centers: The dReDBox project vision"*. Design, Automation and Test in Europe Conference and Exhibition (DATE), pg: 690-695, 2016.

[11] S.L.Lu, *"Evaluation Methods of Computer Memory System"*. 2015 International Symposium on VLSI Design, Automation and Test(VLSI-DAT), pg: 1-4, April 2015

[12] S.C.Woo M.Ohara E.Torrie J.P.Singh A.Gupta *"The Splash-2 Programs: Characterization and Methodological Considerations"*. In Proceedings of the 22nd Annual International Symposium on Computer Architecture, pg: 24-36, June 1995.

[13] C.Sakalis, C.Leonardsson, S.Kaxiras, A.Ros, *"Splash-3: A Properly Synchronized Benchmark Suite for Contemporary Research"*. 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pg: 101-111, April 2016.

[14] C.Bienia, S.Kumar, J.P.Singh, K.Li, *"The PARSEC benchmark suite: Characterization and architectural implications"*. Parallel Architectures and Compilation Techniques (PACT) International Conference, October 2008.

[15] Micron Technology, Inc
`https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr4/4gb_ddr4_sdram.pdf`

[16] Cachegrind: a cache and branch-prediction profiler
`http://valgrind.org/docs/manual/cg-manual.html`

[17] OProfile
`https://perf.wiki.kernel.org/index.php/Main_Page`

[18] perf: Linux profiling with performance counters
`http://oprofile.sourceforge.net/about/`

[19] Y.Kim, W.Yang, O.Mutlu, *"Ramulator: A Fast and Extensible DRAM Simulator"*. IEEE Computer Architecture Letters, Volume: 15, Issue: 1, pg: 45-49, March 2015.

[20] J.D.Leidel, Y.Chen, *"HMC-Sim-2.0: A Simulation Platform for Exploring Custom Memory Cube Operations"*. 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pg: 621-630, August 2016.

[21] D.Jeon, K.Chung, *"CasHMC: A Cycle-accurate Simulator for Hybrid Memory Cube"*. IEEE Computer Architecture Letters, Volume: PP, Issue: 99, August 2016.