

Computer Architecture Simulation on GPU

John Byron Buitrago Paniagua

Seleccione tipo de documento para optar al título de Doctor en Ingeniería Electrónica y de Computación

Asesor
Fredy Alexander Rivera Vélez
Asesor
Ricardo Andrés Velásquez Vélez

Universidad de Antioquia
Facultad de Ingeniería
Doctorado en Ingeniería Electrónica y de Computación
Medellín, Antioquia, Colombia
2024

Cita	(Buitrago, 2024)
Referencia **Estilo APA 7 (2020)**	Buitrago, J. (2024). *Computer Architecture Simulation on GPU* [Tesis doctoral]. Universidad de Antioquia, Medellín, Colombia.



Doctorado en Ingeniería Electrónica y de Computación

Grupo de Investigación Sistemas Embebidos e Inteligencia Computacional (SISTEMIC).

Centro de Investigación Ambientales y de Ingeniería (CIA).





Centro de Documentación de Ingeniería (CENDOI)

Repositorio Institucional: http://bibliotecadigital.udea.edu.co

Universidad de Antioquia - www.udea.edu.co

El contenido de esta obra corresponde al derecho de expresión de los autores y no compromete el pensamiento institucional de la Universidad de Antioquia ni desata su responsabilidad frente a terceros. Los autores asumen la responsabilidad por los derechos de autor y conexos.

UNIVERSIDAD DE ANTIOQUIA

DOCTORAL THESIS

Computer Architecture Simulation on GPUs

Author:
John Byron
BUITRAGO PANIAGUA

Supervisors:
Dr. Fredy Alexander
RIVERA VÉLEZ
Dr. Ricardo Andrés
VELÁSQUEZ VÉLEZ

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

at the

Facultad de Ingeniería
Doctorado en Ingeniería Electrónica y de Computación

September 3, 2023

Declaration of Authorship

I, John Byron
BUITRAGO PANIAGUA, declare that this thesis titled "Computer Architecture Simulation on GPUs" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

Abstract

Due to the power wall in computer architectures [33], processor manufacturers have adopted a strategy of increasing the number of cores on modern computers to enhance their performance and reduce power consumption [100][93][21]. As a result, research and innovation are now focused on integrating more cores on a single chip and improving communication and memory systems, resulting in a complex design space for these architectures [93][110]. Due to processor complexity outpacing processor performance, computer architecture simulators tend to slow down over time [28][36]. This fact necessitates more advanced simulation strategies and tools. Computer architects have proposed various techniques to speed up their simulators by reducing their workload and separately exploiting their parallelism [89] [41] [104] [15] [54] [29] [55] [34].

This work aims to accelerate a detailed simulation of an uncore in a multicore architecture that supports multithreading workloads. The work proposes a methodology that integrates the core models technique [16] that abstracts the core behavior and reduces the time needed to simulate the cores. It also incorporates techniques that leverage the parallel capacity of GPU platforms to speed up the simulation. The methodology includes establishing a core model, a target uncore, their parallelization process, and their interaction. Additionally, it involves simulator task distribution and optimizing GPU process runtime.

The proposed methodology is evaluated through a case study, testing simulations of architectures from 1 to 128 cores to evaluate the speedup scaling and estimate the simulation's accuracy. The reference model consists of a sequential CPU GEM5-based version. The simulation cycles and the number of misses for each cache in the architecture are compared for accuracy. The experimental results show that increasing the number of cores to be simulated in the architecture increases our GPU-based simulation's speed compared to the CPU version. The parallel simulator shows an acceleration starting at 64

cores. In large part of the results, the accuracy for the simulation cycles has a relative error lower than 10%. The accuracy for the number of misses is a consistent metric through all cache levels. The relative error in most cases is lower than 5%. However, performance and accuracy metrics were dependent on workload type. Some simulations presented relative errors of around 40% for the simulation cycles, and some benchmarks did not reach any speedup at 128 cores.

Acknowledgements

I want to express my sincere gratitude to my advisors, Ricardo Andrés Velásquez and Fredy Alexander Rivera, for their unwavering support, motivation, invaluable guidance, and incredible patience. I'm deeply appreciative of the Sistemic group and their dedicated teachers, as well as CFAED and the backing from Professor Jerónimo Castrillón during my research internship. I'd also like to thank Mónica Vargas for her enlightening contributions that significantly enriched the culmination of this thesis.

I extend my heartfelt thanks to Mr. Clemens, Susanne, Rene, Rosa, María, and Thomas—extraordinary individuals I had the privilege of meeting. They not only shared extraordinary moments with me but also provided consistent support during the most challenging times. To my university friends, whose stories and guidance left a lasting impression on this journey, your camaraderie has been invaluable. I'm also grateful to all my friends and relatives who remained attentive and encouraged me not to give up.

My deepest appreciation is reserved for my mother and grandmother, who have selflessly given their all to support me. I'm infinitely grateful to life for blessing me with their presence and showering me with abundant love. Lastly, I extend my profound love to my sons, Juanjo and Nico, who inspire me daily, and to my wife, Marcela, for her unwavering love, support, motivation, and great patience.

This work was supported by COLCIENCIAS "Convocatoria de Doctorados Nacionales 727 del 2015".

Contents

Declaration of Authorship iii

Abstract v

Acknowledgements vii

Contents ix

List of Figures xiii

List of Tables xv

1 Introduction 1

1.1 Problem Statement 1
1.2 Hypothesis 2
1.3 Objectives 2
1.3.1 General Objective 2
1.3.2 Specific Objectives 3
1.4 Methodology 3
1.5 Contributions of this Work 5
1.6 Outline of this Work 6

2 Background and State of the Art 7

2.1 Computer Architecture Simulation 7
2.1.1 Functional Simulators vs. Timing Simulators 8
2.1.2 Application-Level Simulators vs. Full-System Simulators 9
2.1.3 Trace-driven Simulation vs. Execution-driven Simulation 9
2.2 Improving Simulation's Performance 10
2.2.1 Reducing the Amount of Simulation Work 10
2.2.1.1 Sampled Simulation 10
2.2.1.2 Statistical Simulation 11

2.2.1.3 Aproximated Simulation 11
2.2.2 Exploiting Simulator's Parallelism 14
2.2.2.1 Distributing Simulations across Parallel Machines 14
2.2.2.2 Parallel Simulation in a Multithreading Machine 15
2.2.2.3 FPGA-Accelerated Simulation 15
2.2.2.4 GPU-Based Simulation 16
2.3 The GPU Architecture and the CUDA Parallel Programming Model 18
2.3.1 Single Instruction Multiple Threads (SIMT) Structure . 18
2.3.2 CUDA Parallel Programming 19
2.3.3 Memory Hierarchy in a GPU 20

3 Designing our Computer Architecture Simulator 21
3.1 Computer Architecture and its Simulation Strategy 21
3.2 Our Simulation Approach: Simulator Block Diagram 22
3.2.1 Core Model Simulator 23
3.3 Tasks Distribution between CPU and GPU 25
3.4 Optimizing GPU Processes Runtime 27
3.5 Summary 30

4 Implementing a Core Model into the GPU 33
4.1 Trace-Driven Core Models 33
4.2 Synchrotrace Core Model 35
4.2.1 Trace Generation 35
4.2.2 Synchrotrace Simulation 36
4.3 Modifying Synchrotrace Model 38
4.4 Implementing a Parallelized Version of the Core Model 43
4.4.1 Description of the Trace Events Data Structure 43
4.4.2 Event Processing through GPU Kernels 44
4.4.3 Implementing the Core Model Timing Controller . . . 46
4.5 Summary 46

5 Parallelizing the Uncore Simulator for a GPU 49
5.1 Uncore Architecture 49
5.1.1 Cache Memories 50
5.1.1.1 Managing Cache Requests 52

5.1.1.2 Managing Cache Responses 54
5.1.2 Communication Bus 55
5.1.3 MOESI Snoop Process 56
5.1.4 Computational Cost Analysis for the Target Architecture simulation 59
5.2 Parallelizing the Uncore Simulation 63
5.2.1 Data Parallelism 63
5.2.1.1 Coarse-Grained Data Parallelism: Nodes . . . 63
5.2.2 Task Parallelism 63
5.2.2.1 Pipelined Structural Levels in Target Architecture 63
5.2.3 Kernel Parallelism Dimensionality (Number of Threads) 63
5.2.4 Computational Cost Analysis for the Parallelized Architecture Simulation 66
5.3 Summary 69

6 Experimental Results **71**
6.1 Experimental Setup 71
6.2 Evaluation of GPU-based Simulator 74
6.2.1 Scaling the GPU-based Simulation Speedup 74
6.2.2 Analysis of Speedup Results 77
6.2.3 Accuracy in the GPU-Based Simulator 79

7 Conclusions and Future Work **83**
7.1 Future work 85

A CUDA Kernels Algorithms **87**
A.1 Core model Kernels 87
A.2 Cache memory Kernels 90
A.3 Shared bus Kernels 94

Bibliography **95**

List of Figures

2.1 Simulator's design dimensions 8
2.2 Taxonomy of methodologies 10
2.3 Core and uncore systems 13
2.4 SIMD taxonomy 18
2.5 Thread hierarchy 19
2.6 Memory hierarchy and thread hierarchy 20

3.1 Target computer architecture 22
3.2 Block diagram of computer architecture simulator 23
3.3 Core model simulation 24
3.4 CPU-GPU process interaction 26
3.5 FSM to control trace events CPU-GPU transfer 26
3.6 Sequential steps in a computer architecture simulator 28
3.7 Pipelined steps in our computer architecture simulator 29
3.8 Double buffer arrangement 29
3.9 Message flow in every simulation cycle 30
3.10 Cuda graph for our simulation model 31

4.1 Synchrotrace events types 36
4.3 Synchrotrace playback process 37
4.4 Breaking down Synchrotrace events 39
4.5 Regular structure to store GPU-based events 43
4.6 Data and task parallelism in the core model simulator 44
4.7 Simulation timing control 46

5.1 Target computer architecture with coherence mechanism . . . 50
5.2 Fields in a cache line 50
5.3 Array of lines in a cache 51
5.4 Line sets in an 8-way associativity cache 51
5.5 Array of cache lines at a specific level 52
5.6 Interblock communication channels 52

5.7 Request message structure 53
5.8 Detailed description of the process to determine hit/miss . . . 53
5.9 Response message structure 54
5.10 Detailed description of the placement process 55
5.11 Communication bus subprocesses 55
5.12 State machine for L3 access queue manager 56
5.13 Meaning of states in the MOESI coherence protocol 57
5.14 MOESI coherence protocol when caches process a request . . . 57
5.15 MOESI coherence protocol in the snooping process 58
5.16 Communication between snooping process and caches 59
5.17 Processing flow for a requested data 60
5.18 Kernel parallelism in L1 cache request 64
5.19 Parallelism grade in the target computer architecture 65
5.20 Comparison between sequential and parallel model 69

6.1 GPU simulation scaling 75
6.2 GPU simulation scaling until 128 threads 76
6.3 Theoretical and experimental speedup comparison 76
6.4 MIPS comparison 78
6.5 SimTicks detailed accuracy 79
6.6 Detailed misses cache Accuracy 81

List of Tables

4.1 Comparison of Trace-driven models 34

5.1 Uncore kernel list, dimensionality, and parameters 64

6.1 Configuration values 72
6.2 Splash2 benchmarks description [109] [87] 73
6.3 List of excluded benchmarks 74
6.4 Traffic and Synchonization metrics in Splash2 benchmarks . . 77
6.5 MIPS in state-of-the-art sequential simulators for one-core . . 78

Chapter 1

Introduction

Since the first decade of this century, the exploitation of Instruction Level Parallelism (ILP) has reached its limits, and the power wall is limiting the clock frequency on modern computers [21]. As a result, increasing the number of cores is becoming increasingly important to improve computer performance and reduce power consumption [93].

As proof of this, AMD and Intel are in fierce competition in the general-purpose and server processor markets, seeking to offer processors with higher cores. The launch of AMD's ZEN architecture and Ryzen and EPIC product families sparked this competition [100] [93]. This trend highlights the shift in modern processor research and innovation, now focusing on integrating more cores on a single chip and improving communication and memory systems (the uncore) instead of seeking higher ILP innovations. Thus, computer architects require more advanced simulation strategies and tools to handle the complex design spaces of modern processors [70] [110].

1.1 Problem Statement

Computer architecture simulators tend to become slower over time as processor complexity increases faster than performance [37] [28]. Nevertheless, computer architects continue to propose strategies to enhance simulator performance. One such strategy involves utilizing hardware accelerators based on FPGAs and GPUs to speed up computer architecture simulations [46] [20] [85] [50] [91] [97] [82] [98]. Although this approach has resulted in faster simulation speeds, many simulators use low-level detailed microarchitectural models that do not scale well with parallel programming. Additionally, using accelerators as simulation platforms has not been widely adopted due

to the complexity of parallelizing a simulator and programming these devices. Interestingly, most modern supercomputers offer significant computing power via GPUs and, in some cases, even FPGAs [23] [58] [75], which could be utilized to speed up computer architecture simulations.

On the other hand, computer architecture scientists have also proposed strategies to reduce the simulator's computational load [29] [34]. One approach is reducing the number of simulated instructions. Another strategy involves replacing certain simulator parts with simpler, abstract models that reduce complexity and speed up operation [89][41][104][15][54][55]. Specifically, core models replace the complex simulation of the core with a simpler abstract model, focusing only on the core's interaction with the uncore. This simplification of the simulation model enables a more straightforward evaluation of new design ideas in the uncore, which is the current focus of architectural optimization.

A computer architect must use the entire arsenal of acceleration strategies to achieve efficient simulation speeds. Combining the core models strategy and hardware acceleration with GPUs can provide faster simulation speeds in studies where the research focuses on the uncore. In addition, the uncore simulation model is also considerably more regular and repetitive, which can facilitate its implementation on a GPU. We did not find any evidence in the literature about a simulation strategy of this kind.

1.2 Hypothesis

We can define an efficient strategy for combining core models, a detailed uncore simulation model, and GPU computation. This combination leads to the development of faster multicore simulators without affecting their accuracy and profit from the current supercomputer infrastructure.

1.3 Objectives

1.3.1 General Objective

To propose a methodology that combines core models, a detailed uncore description, and GPU to enable the implementation of accurate and fast architecture simulators.

1.3.2 Specific Objectives

1. To establish a methodology to integrate a core model and a parallel uncore simulator.
2. To choose a fast and accurate core model technique that supports multithread applications with a feasible GPU implementation.
3. To model an uncore in a parallel fashion to be mapped onto a GPU.
4. To implement the parallel simulation model of the uncore for its execution on GPUs.
5. To evaluate the established methodology through a case study.

1.4 Methodology

This section describes the methodology that allows us to achieve our objectives.

Our first task was to select a core model suitable for simulating multicore architectures. This model should be capable of handling real-world workloads, allowing us to evaluate the simulation of architectures under realistic conditions. We reviewed existing core models and found a few that met our criteria of having open-source code and support for multithreading workloads. We identified Sniper and Synchrotrace as models that fit these requirements. However, there needed to be a standardized method for measuring the reported metrics, making it difficult to compare these simulators' quantitative features, such as accuracy or speedup.

Despite Sniper's popularity in the computer architecture community, we encountered difficulties using it as our core model. While it benefits simulation users, it lacked sufficient documentation to modify its code, which was crucial for implementing our proposed simulation approach. On the other hand, Synchrotrace had adequate development documentation to support our project, despite its limitation of modeling an essential in-order core. As there were no better alternatives, we ultimately chose Synchrotrace as our core model.

With Synchrotrace selected as the core model, we evaluated the options for integrating it with the uncore simulator. There are two basic alternatives: 1)

to keep the core model on the CPU or 2) to move the core model processing onto the GPU. To ensure smooth communication between the CPU and GPU, we chose the latter and developed a parallel version of Synchrotrace to run on the GPU. We modified the original Synchrotrace algorithm and its associated traces to ease GPU programming increasing regularity and suitability. We implemented the algorithm using CUDA.

At this point, we moved on to designing and implementing the uncore simulator. Our first step was to determine the composition of the uncore, which included a conventional multicore processor with three levels of the memory hierarchy (private L1 and L2 cache memories and shared L3), an interconnection bus, a cache coherence protocol, and main memory. Next, we opted for an event-driven simulation model more suitable for accelerating the simulation process than a cycle-driven approach. We used CUDA to develop the uncore simulator. We map hardware structures to CUDA kernels and leverage the multiple instances of hardware structures to increase parallelism. We also faced the challenge of distributing the work among the available CUDA cores by defining suitable data structures for the simulator's temporal information.

The integration progress between the core model and uncore simulator was straightforward once they were both CUDA programs. The core model generates the memory requests to process by the uncore simulator. The uncore simulator notifies the core model of each memory request's latency. However, additional programming was necessary to manage the simulation loop, the simulation time flow, and the statistics generated by the simulator. Debugging the full simulator was time-consuming, and functional testing involved simulating real workloads (billions of instructions), which required significant effort and multiple tests to identify and correct errors.

After thoroughly testing the simulator, we evaluated the speedup by comparing it to the reference implementation of Synchrotrace integrated with the uncore provided by the GEM5 simulator. Although we adjusted our simulator model and configured the GEM5 uncore to make the two models as similar as possible, there were still some differences between the two models. We used the Splash2 benchmark suite [87][109] as our multithreaded workload and measured the execution time and simulated time for both simulators for each benchmark. We also analyzed how these times changed with different

core counts. Our experimental results aimed to showcase the performance of our simulator compared to the reference Synchrotrace-GEM5 simulator and gave insight into the potential of our simulation strategy. Despite not having an identical reference model with the same functional characteristics as our parallel simulator, we compared the simulation cycles and cache misses between both simulators to estimate the coherence in simulation results.

The purpose of our experimental results was to demonstrate the acceleration of our simulator compared to the reference Synchrotrace-GEM5 simulator and to provide an understanding of the potential of our simulation strategy. Although we lacked an identical reference model with the same functional characteristics as our parallel simulator, we still compared simulation cycles and cache misses between both simulators to assess the consistency of simulation outcomes.

1.5 Contributions of this Work

The dissertation's contributions are as follows:

1. We introduce a novel methodology for implementing computer architecture simulators that particularly emphasizes the uncore. Our approach is based on a combination of approximated simulations of the cores, a detailed model of the uncore, and parallelization techniques through GPUs.

2. We developed a modified parallel Synchrotrace model specifically designed to handle simpler and more regular events, making it well-suited for execution on GPUs. This parallel model enables easy integration with a parallelized uncore.

3. We developed a detailed parallel model of the uncore, including the most important components in a modern multicore processor memory hierarchy. We applied several parallelization strategies to minimize simulation time.

4. We developed an analytical performance model for our simulator's execution on the GPU and CPU, which helped us to estimate the theoretical speedup and identify the limitations of our model. The model predicted the performance of our simulator with reasonable accuracy

compared to the actual results. This contribution enabled us to optimize our simulator's performance and improve its accuracy in predicting the execution time.

5. Our methodology constitutes a notable contribution to computer architecture simulation as it demonstrates the use of GPUs as host platforms for simulating detailed uncores in architectures with hundreds of cores and beyond. This simulation methodology represents a valuable addition to the toolbox of computer architecture simulation.

1.6 Outline of this Work

The rest of this thesis is structured as follows: Chapter 2 provides background and an overview of state of the art in computer architecture simulation, GPU architecture, and the CUDA parallel programming model. Chapter 3 outlines a methodology for integrating a core model and an uncore for a parallel GPU-based simulator. Chapter 4 explains the selection of a core model and its implementation methodology in a GPU platform. Chapter 5 describes the uncore's components and presents a detailed version's implementation in a GPU, along with an analysis of the run time complexity of the uncore simulation and the ideal speedup of the parallel simulator. Chapter 6 presents a case study that evaluates the proposed methodology, including the experimental setup, speedup, and accuracy results. Finally, Chapter 7 summarizes the conclusions and outlines future work.

Chapter 2

Background and State of the Art

This chapter presents a comprehensive review of the current state-of-the-art and existing methods used to improve the performance of computer architecture simulation. It also describes the GPU architecture and outlines the CUDA parallel programming model.

2.1 Computer Architecture Simulation

The objective of computer architecture simulators is to predict the behavior of computer platforms [44]. A simulator generally consists of an instruction emulator that executes a workload and a timing simulation engine that simulates the execution of instructions on a hypothetical machine. One run of a simulation model only provides a "snapshot" of the system behavior; then, several simulations run on different benchmarks must be completed, and the results must be carefully analyzed to obtain a global picture of the system [44, 90]. Evaluating the impact on performance due to changes in a system parameter will require repeating the simulation runs on all the benchmarks for every possible parameter value [29], implicating ample time to get performance results of the system.

A series of orthogonally independent design dimensions should be explored when looking at a specific computer simulator design. The design dimensions are scope, detail, and input [4]. The following sections explain these dimensions, and Figure 2.1 summarizes them.

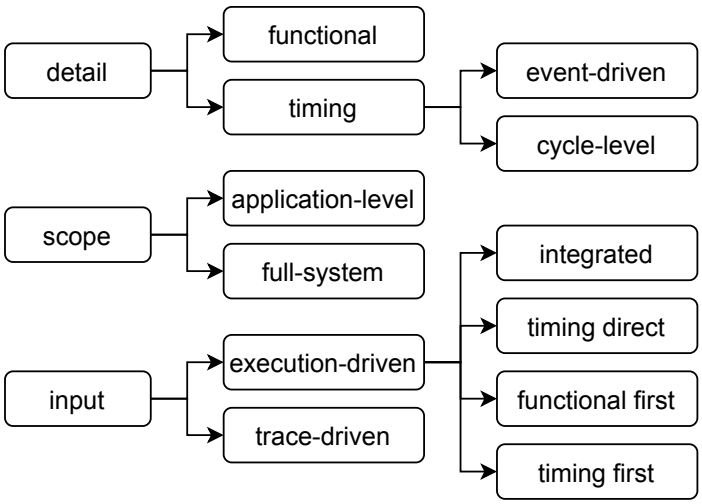


FIGURE 2.1: Simulator's design dimensions

2.1.1 Functional Simulators vs. Timing Simulators

A functional simulator emulates the instruction execution of a target architecture and does not care about the micro-architectural details or instruction timing [5] [4]. These tools help validate architecture's correctness rather than evaluate its performance [29]. Virtual platforms such as HVP [17] and the work of Weinstock et al. [106] are examples of functional-only simulators. These works abstract away the hardware details and aim at early MPSoC software development, thus enabling highly reusable code. TOTAL-PROF [32] is a work that exploits the high speed of functional simulation to achieve fast and complete profiling in applications targeted at specific or general computer architecture. Other examples of these simulator types are Cotson [7], Simnow [9], Simics [64], and the 'sim-safe' and 'sim-fast' SimpleScalar's models [8].

Timing simulators execute the instructions and provide detailed information about the accurate timing/performance of a simulated complete computer system or only a part of it. Timing simulators enable measuring performance figures such as the number of cycles, power, reliability, and energy efficiency. Keeping track of such details slows timing simulators and implies many computational resources compared to functional simulators. There are two approaches to developing timing simulators. Simulators that model cycle-by-cycle the target computer are known as cycle-accurate simulators, such as *gem5+rtl* [60], Tejas [90], Parade [25], Marss86 [80], PTLsim [111], Zesto [59] and the work in [112]. Simulators that only keep track of events are called

event-driven simulators, which are faster but less accurate [4], such as works reported in [66], [61], and [51]. Some simulators combine both cycle-level and event-driven approaches. Examples of these simulators are McSimA+ [3] and SimFlex [107].

2.1.2 Application-Level Simulators vs. Full-System Simulators

Application-level simulators run only target programs instead of an entire target operating system (OS). They do not simulate what happens upon an operating system call or interrupt. The most common approach is to ignore interrupts and emulate the effect of system calls [4]. Application-level simulators are usually less complex but may show inaccuracies because they do not account for OS effects. Examples of application-level simulators are Sniper [15], Zesto [59], McSimA+ [3], and ZSim [88].

Unlike application-level simulators, full-system simulators simulate an entire computer system. Hence, a complete software stack can run on the simulator, including an entire OS [4, 29]. Examples of full-system simulators are Tejas [90], GEM5 [11], Marss86 [80], and PTLsim [111].

2.1.3 Trace-driven Simulation vs. Execution-driven Simulation

Trace-driven simulators take program instructions and memory access traces from a benchmark. The simulator injects traces into a detailed microarchitecture timing simulator [29]. In general, trace-driven simulators separate functional simulation from timing simulation. The main advantage of trace-driven simulators is that they execute the application once and use traces to evaluate the performance of many architecture configurations [29].

Some disadvantages of trace-driven simulation are the need to store the trace files and the impossibility of modeling the effects along mispredicted program paths [45]. Another relevant disadvantage is the difficulty of modeling the impact of the microarchitecture on inter-thread ordering when simulating multi-thread workloads [4]. Several techniques, such as sampling, have been proposed to reduce the trace size and improve the performance of this approach [13].

Execution-driven simulators combine a functional model with a timing model. This kind of simulator does not have the disadvantages of trace-driven simulators and is the de-facto simulation approach [29]. However, the higher accuracy of execution-driven simulators leads to increased development time and a lower simulation speed [104] [29]. Examples of execution-driven simulators are GEM5 [11], SimpleScalar [8], and PTLsim [111]. According to the interaction between functional and timing models, these simulators are classified as integrated, timing-directed, functional-first, and timing-first [29, 68].

2.2 Improving Simulation's Performance

There are two approaches for improving simulation performance: (i) Reducing the amount of work either increasing efficiency or eliminating unnecessary work; (ii) Exploiting the parallelism of the simulator [104]. Figure 2.2 summarizes the main techniques used to accelerate simulations.

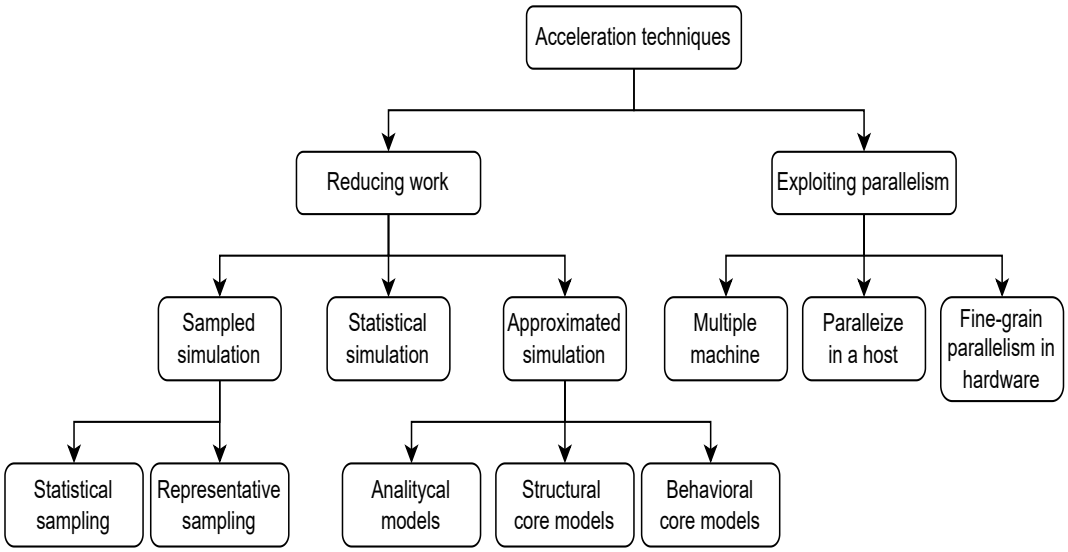


FIGURE 2.2: Taxonomy of accelerating techniques

2.2.1 Reducing the Amount of Simulation Work

2.2.1.1 Sampled Simulation

Sampled simulation is a technique that reduces execution time by simulating only a fraction (samples) of a longer benchmark [29][26]. In this context, a sample is a contiguous interval of dynamic instructions during program

execution. To be accurate and fast, sampled simulation must: (i) select a sample that is representative of the entire simulation, (ii) set up a sampling unit's architecture starting state (register and memory state), and (iii) estimate the microarchitecture state at the beginning of a sample (e.g., branch predictor and processor structures) [29]. TQSIM [47] uses sampled simulation techniques to achieve a speedup between 41× and 135× concerning a cycle-accurate GEM5 simulator.

2.2.1.2 Statistical Simulation

Statistical simulation reduces the complexity of the simulator generating small synthetic workloads that are representative of long-running benchmarks [38]. This technique collects several program characteristics in the form of distributions. Then, it generates a synthetic trace shorter than the original program, but the simulation performance results represent the whole actual program. A statistical simulation aims to reduce simulation time and serves as a fast-design space exploration technique [29].

In [34], a statistical simulation technique for chip multiprocessors running multi-program workloads by modeling shared resources, such as the last-level caches, is proposed. This approach has an average error of less than 5.5% and simulation speedups of 40× to 70× compared to a cycle-accurate simulation of 100 million instruction traces. Results of statistical simulation experiments show that this technique is helpful for early design exploration [43] [35].

2.2.1.3 Aproximated Simulation

Sampled and statistical simulations reduce the program length to obtain faster measurements, i.e., lessen the simulator's input. An orthogonal approach involves reducing the complexity of the simulator. Simulator complexity reduction lessens the amount of modeled work performed for each input, i.e., it may reduce the simulator's effort given a fixed input or remove the need for a simulation entirely. Approximated simulators raise the level of abstraction in simulation and trade accuracy for simulation speed. They are usually applied for design space exploration and, more generally, for situations where the slowness and the low scalability of cycle-accurate simulators limit their usefulness [104].

Analytical models, which use mathematical equations to estimate the performance of a computer architecture based on micro-architectural parameters, are employed in the approximate simulation approach. Various researchers have proposed such models. Lee et al. [52] [53] have developed and validated regression models to assess power and performance in multiprocessor architecture. Van den Steen et al. [96] have introduced an analytical performance and power model for superscalar out-of-order processors based on microarchitecture-independent application profiles.

In addition, structural core models are used to simulate the internal behavior of the core microarchitecture, with varying levels of accuracy based on the number of modeled parameters. By modeling only "first order" parameters, structural models can speed up simulations of superscalar processors while still capturing the most significant performance impacts. One such model is Sniper [15], an open-source, parallel, and scalable multi-core simulator that simulates computer systems with multiple processor cores and large caches. Sniper maintains an instruction interval for each simulated core, fed by a functional simulator. The interval simulation approach implemented in Sniper models the timing for each core, abstracting core performance by driving the timing simulation without detailed tracking of individual instructions through the core's pipeline stages. Another example of a structural core model is In-N-Out [54], which employs an approximate simulation method to replicate dynamic execution behavior in superscalar processors. This approach uses filtered instead of full traces to reduce the storage space required for preparation and shorten simulation times.

On the other hand, behavioral core modeling, or the core model technique, is a possible way to trade off accuracy for simulation speed in situations where the focus of the study is not the core itself but what is outside the core, i.e., the uncore [21]. This document uses the term "uncore" to refer to the cache memories and shared resources, including shared memory and on-chip interconnect. Figure 2.3 shows a multi-core system's core and uncore components. This system presents a set of cores with private caches and shared resources: an interconnection bus, a shared cache, and the main memory. In this case, the figure represents the cores with a primary pipeline.

Behavioral core models are abstractions of detailed core functionality. Some simulators can connect core models to a cycle-accurate uncore model and

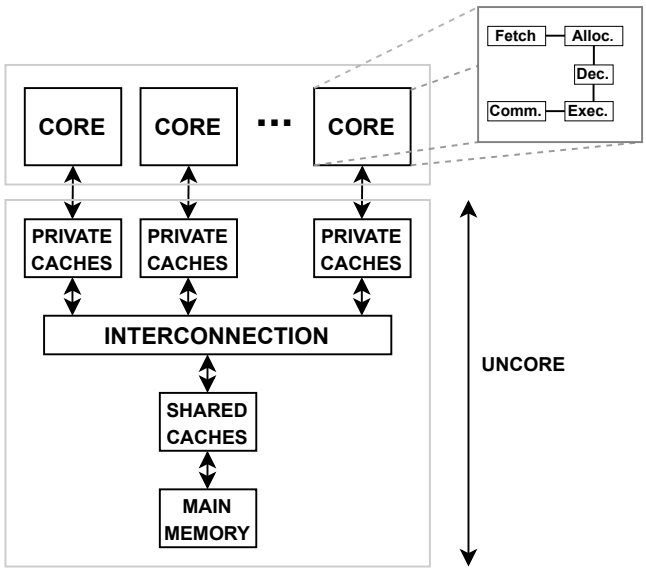


FIGURE 2.3: Core and uncore systems

emulate the core's external behavior as a black box emitting requests to the uncore at certain times. This technique uses annotated traces to represent the core and the uncore interaction. Whereas other techniques use an entire stream of instructions, core models use filtered annotated traces [15] [41] [54] [55] [73] [104]. There is a trade-off between reducing the amount of work and accuracy. However, approximated measurements are enough for specific fields, such as early design space exploration [74].

BADCO [104] defines behavioral models for modern superscalar cores and can predict how their performance changes when the uncore changes. This model uses a behavioral method to find dependencies among requests to the uncore. BADCO's simulation flow has three main phases: trace generation, model building, and model simulation phase. When BADCO core models have been built for single-thread benchmarks, the core models can be easily combined to simulate a multi-core running of several independent threads simultaneously. This approach reports a simulation cycle error of less than 10% and a speedup between one and two orders of magnitude regarding the Zesto cycle-accurate simulator [59]. BADCO helps explore the design space of the uncore or studying multiprogramming (independent threads simultaneously) workloads. However, it does not support multi-threading (dependent threads simultaneously) workloads.

Lee *et al.* [55] present a strategy that uses storage-efficient filtered traces, which carry enough information to compute cycles between trace items at

simulation time. This work focuses on how a superscalar processor's out-of-order instruction execution capability, as a function of the Re-Order Buffer (ROB) size, affects the trace-driven simulation methodology. This approach achieves simulation speedups of 30× and an error of 4.8% for cache misses when compared with a detailed execution-driven simulator.

Jagtap *et al.* [41] propose elastic traces, extracting data and memory access dependencies instrumenting an out-of-order processor. Elastic traces support modeling an out-of-order core with a configurable load/store buffer and ROB size and recording actual physical addresses. This work provides a trace player and a SystemC TLM 2.0 interface, enabling integration with other tools. Elastic traces achieve a speedup close to 8× compared to a cycle-accurate simulation and present an average error in the execution time of 7% in standalone mode.

SynchroTrace [73] is a two-step methodology for trace-based simulation of multithreaded applications. This approach can model the inherent non-determinism in the execution of multithreaded programs [6]. SynchroTrace captures and embeds synchronization events in each trace and tracks dependencies between traces. This model identifies, within the traces, events of different types: computation, synchronization, and communication. This approach presents a speedup of 13.4× when compared with the GEM5 [11] full-system simulator.

2.2.2 Exploiting Simulator's Parallelism

Four approaches use parallelism to speed up simulation [5] [29] [78]: (i) to distribute multiple simulations running across a cluster of computational machines, (ii) to parallelize the simulator to benefit from parallelism in a multithreading machine, (iii) to exploit fine-grain parallelism for mapping (parts of) a simulator on reconfigurable hardware (i.e., FPGAs), and recently (iv) to simulate many target cores that are mapped to many hardware-threads available on highly parallel GPU platforms.

2.2.2.1 Distributing Simulations across Parallel Machines

This approach parallelizes multiple simulations loading individual simulator instances in each machine. This approach is helpful for design space exploration, where multiple design points must be simulated. However, this

approach does not reduce the latency of obtaining a single simulation result. This approach's parallelism level depends on the number of design points. In tuning parameters of computer design, designers could prefer individual simulations that finish in a few minutes or hours over tens or hundreds of results after hundreds of hours of waiting [29]. One example of this approach is Graphite [69], a distributed parallel multi-core simulator infrastructure. This infrastructure can distribute simulator work across a machine cluster to accelerate the simulation. Graphite on 10 Linux machines can simulate target architectures with up to 1,000 cores. For the simulation of Splash2 benchmarks on a target architecture of 32 cores, this simulator achieves slowdowns from 41× to 3930× over native execution. Another work that uses this strategy is Simics [63].

2.2.2.2 Parallel Simulation in a Multithreading Machine

Parallel simulation enables scalability in multi-core processor technology [29]. The basic idea of this approach is to divide the simulator across multiple simulator threads. Then each simulator thread executes part of the work. One possible partition is to map each target core onto a separate simulator thread and have one or more simulator threads simulating the shared resources. A parallel simulator can be implemented as a single address space shared-memory program. Alternatively, the parallel simulator can be a distributed program that uses message passing to communicate among the simulator threads. Examples of this simulator type are Graphite [69], Simics [63], BigSim [113], ZSim [88], Sniper [15], and BARRA [24].

2.2.2.3 FPGA-Accelerated Simulation

This approach to exploit simulator parallelism mainly uses multi-core processors, FPGAs, or a combination. Most of these works use system implementations based on Hardware Description Languages (HDLs). RAMP gold [98] is a cycle-accurate full system simulator on a single Xilinx Virtex-5 FPGA board. It can simulate a 64-core shared-memory target, achieving 50 MIPS on the execution of PARSEC benchmark suite and a 250× speedup compared to the software-based architecture simulator Simics [63] + GEMS [65].

HASIM [82] simulates a shared-memory multi-core system with precise core pipelines, cache hierarchy, and a cycle-accurate on-chip network. This simulator runs on a Xilinx Virtex 5 FPGA in the Nallatech ACP platform [1]. This approach presents a fine-grained time-multiplexing scheme, enabling 16 FPGA cycles to simulate one model cycle that enables a peak performance of 3MHz, multiple orders of magnitude faster than software-only models [30].

HYSIM [97] bases its functionality on RAMP gold. However, the timing model of the memory subsystem is moved from the FPGA to the host, enabling scalability, heterogeneity, and the ability to model different on-chip-network topologies. MASHfifo [91] is a multiple cache simulator in hardware; this work obtains a simulation time speedup of up to 11× compared with the software alternative, CIPARSim [39]. Traces were generated 730× faster on average compared to the software simulator, and cache simulation throughput increased by 6.78×.

Adapnoc [46] is a Network on Chip (NoC) co-simulator that uses the inherent parallelism of FPGAs to speed up orders of magnitude against software simulators. This approach is a configurable cycle-accurate FPGA-based NoC simulator that obtains 53× to 180× speedup against the software version.

MRP [20] is a multi-core simulation methodology that mixes real and pseudo-cores for FPGA-based multi-core simulation. This work proposes and implements several types of pseudo cores and validates their effectiveness by comparing experiments against the corresponding ASIC chip. This approach achieves a speedup of about 800× over GEM5 [11].

One common characteristic of these approaches is their significant synthesis time. Any change in the design parameters implies a considerable increase in the development time of simulators [5]. In addition, the programming complexity of these simulators has become an obstacle to their widespread adoption [94].

2.2.2.4 GPU-Based Simulation

The simulation of many-core computer architectures is inherently parallel because of the vast number of repeated physical units. GPU platforms are a good option for carrying these simulations due to their high computational capacity and the thousands of concurrent processing units in modern GPUs.

Several approaches have exploited the computation capacity of GPUs based on their large number of processing cores [50] [84] [62] [49] [105] [83].

DDGSim [50] is a GPU parallel version of a chip multi-core simulator with distributed cache management. Target is a multi-core system that implements a bufferless mesh interconnection network to connect its cores. It has a centralized directory without a cache coherence mechanism to manage distributed data in each node. Each node has a core and a private L1 and L2 cache with partial slices of total data in the network. Input to the simulator is a trace, which provides only memory access addresses from the core. This simulator generates traces using the Multi2sim simulator [102] from SPEC OMP 2001 benchmark. Simulation carries out the work in three phases: (i) L1 and L2 cache access in each core, (ii) sorting requests in each router, and (iii) transferring requests to the next router. The model assigns each core and router functionality to one GPU CUDA thread. The parallelized simulator was compared against a serial version implemented by the same authors. They achieve a speedup of 25× over the serial simulation with a target system with 2000 cores. This work implements a simplified, centralized directory without cache coherence protocol and without supporting multithreading applications.

Raghav *et al.* [84] developed a methodology to accelerate the simulation of a many-core coprocessor using GPU platforms. The proposed target architecture is a specific heterogeneous system composed of an ARM v7 general-purpose processor connected to 4096 simple ARM in-order coprocessors interconnected using a rectangular mesh. Each target coprocessor is a node in the mesh network. This node comprises a core, private cache, private and distributed shared memory, and a Network On Chip (NOC) switch. The model maps each simulated node onto a single CUDA thread. A QEMU [10] emulator instance simulates the CPU target, and a GPU platform supports the simulation of coprocessors. This work explores the number of fixed coprocessors scaling in an interconnection network and does not allow a design space exploration of the target system's uncore. The coarse-grained model prevents scaling when internal structural parameters can be modified. Evaluation of this approach consisted of simulating architectures from 128 to 4096 coprocessors with Dhrystone, Fibonacci, Matrix Multiplication (MM), and Normalized Cut Clustering (NCC) kernels. The authors measure Millions of Simulated Instructions per second (S-MIPS) in the simulator and compare

this metric in the only-functional and sequential simulator OVPSIM [99] simulating the same architectures. This work reports a speedup of 2.2×, 1.1×, 15×, and 18× for Dhrystone, Fibonacci, MM, and NCC benchmarks, respectively, when simulating a target architecture with 4096 processors.

2.3 The GPU Architecture and the CUDA Parallel Programming Model

2.3.1 Single Instruction Multiple Threads (SIMT) Structure

Flynn's taxonomy [31] defines Single Instruction Multiple Data (SIMD) as a parallel computer structure that can execute the same instruction on multiple processing units operating on different data, as shown in Figure 2.4.

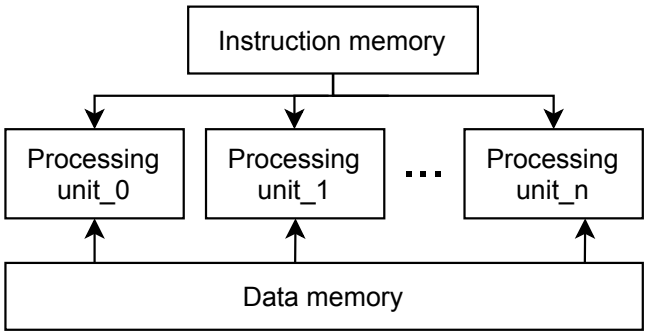


FIGURE 2.4: SIMD taxonomy

An extension of SIMD is the Single Instruction Multiple Threads (SIMT) [67], an essential feature for the General-Purpose Graphics Processing Unit (GP-GPU). Processing units on SIMD are cores in a SIMT architecture passing from multiple data into multiple threads. SIMT handles different control-flow paths. Threads are enabled or disabled according to the path in execution [86]. Control is uniform when all threads in a group follow the same path, and control is divergent when some threads follow different paths. At a specific time, threads with the same control-flow path are running, and others are disabled. Instructions and data are processed only on the active threads. The execution model guarantees that all possible paths in the threads can progress.

2.3.2 CUDA Parallel Programming

The CUDA parallel programming model provides a heterogeneous environment to develop applications with interactions between CPU and GPU processes [77]. CUDA provides a toolkit including libraries, a compiler, and a runtime environment. CUDA extends the C++ programming language with language extensions to include an abstraction of the GPU architecture, memory space allocation, and data transfer between CPU and GPU memories. CUDA defines kernels as the way to express functions. A GPU can concurrently execute the kernel's functionality of n instances in n CUDA threads. CUDA allows configuring kernels to run in blocks of p threads and grids of r blocks, and the same kernel can be called multiple times with different block sizes or grid sizes. Figure 2.6 exhibits the thread hierarchy in CUDA.

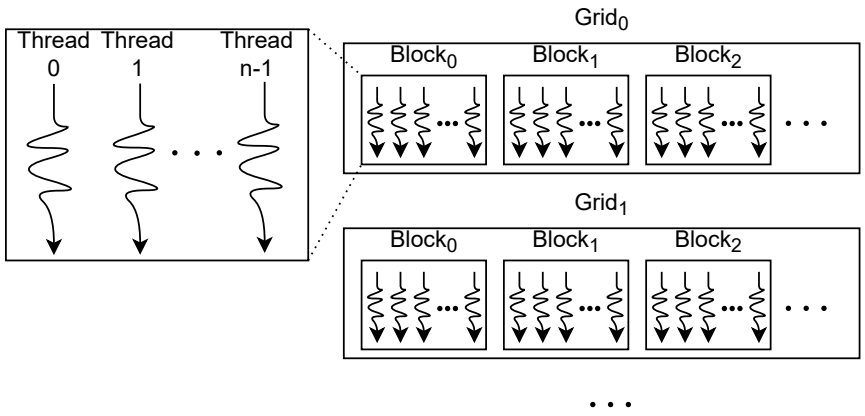


FIGURE 2.5: Thread hierarchy

A CUDA thread runs on an Nvidia-GPU core, called CUDA-core or Streaming Processor (SP) [76]. The SPs are grouped into Streaming Multiprocessors (SM) in multiples of 32 SPs. A warp is a group of 32 threads. When a GPU executes a kernel, the kernel grids and blocks are enumerated and assigned to several computational components. Grids can run on different SMs. On an SM, the threads in a block are also consecutively enumerated and grouped into warps (32 threads) for SIMT execution. Therefore, a set of 32 SPs executes a warp. CUDA can assign multiple thread blocks to an SM corresponding to the same kernel or other kernels. An SM usually has multiple warps to run. The warp scheduler unit manages and controls the warp launching.

2.3.3 Memory Hierarchy in a GPU

Figure 2.6 shows the GPU memory hierarchy and its relation with the GPU thread hierarchy. Each CUDA core corresponding to the CUDA thread level has a private local memory space. This space can reside in SM registers or main memory, depending on programming directives, requested memory space, or availability of memory space. At the block level, there is a shared memory with scoping to all threads in the block. This shared space will be persistent across block run time. At the grid level, the shared memory space is the global memory. Any thread can share information with any thread, in any other block, and any grid. The lifetime of data in the main memory has the application lifetime.

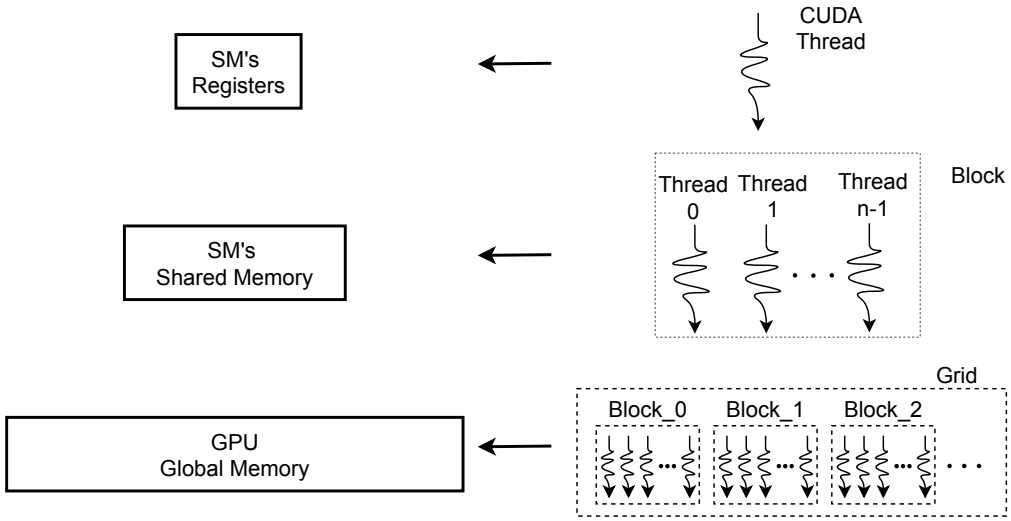


FIGURE 2.6: Memory hierarchy and thread hierarchy

Chapter 3

Designing our Computer Architecture Simulator

Chapter 1 established our main objective for this work: to combine core models with a GPU-based detailed uncore simulation. This chapter elaborates on the design of a computer architecture simulator that fulfills this objective. The chapter highlights the critical design decisions throughout the process. We start defining our target architecture platform to simulate and simulation strategy. Then, we present a simulator's block diagram describing its most essential simulation components. We then introduce each component and its interactions with others and discuss our significant design choices.

3.1 Computer Architecture and its Simulation Strategy

Our target computer architecture is a conventional multicore system (cores and uncore), as Figure 3.1 shows. Although other works establish their uncore with only one private cache level or without private caches [89] [74] [47] [14] [104], our uncore includes private L1 and L2 caches, a communication bus, a shared L3 cache, and the main memory with constant access latency. The architecture includes a coherence protocol to support multithreaded workloads [71]. This architecture can include n cores and the same number of private L1 and L2 caches.

From the simulation model perspective, we implemented an event-driven approach (See Section 2.1.3) to simulate the proposed architecture. Although this approach is less accurate than cycle-accurate, it has a greater simulation

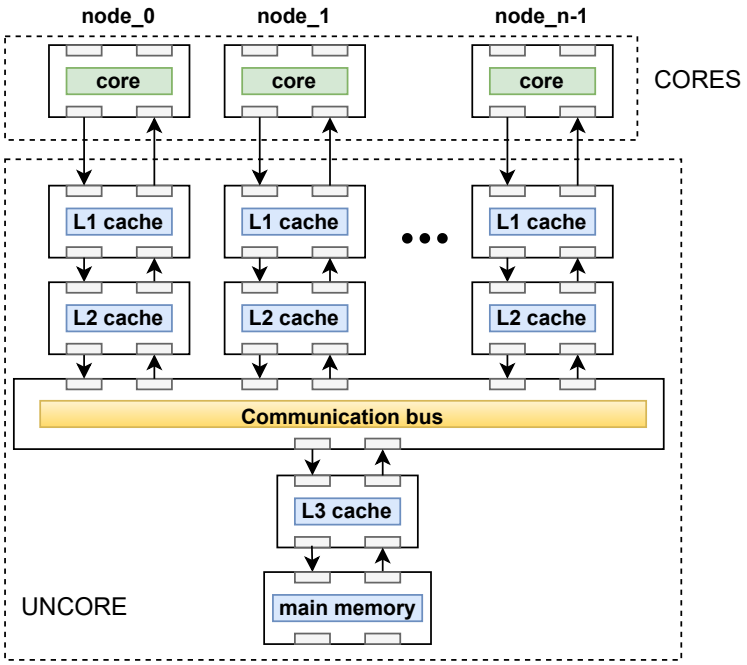


FIGURE 3.1: Target computer architecture

speed. We use core models to simulate the cores. To simulate the uncore in an event-driven mode, we model the uncore as elements that propagate events (memory requests and responses) into adjacent elements. When the uncore finds the requested data, it updates all concerned components and returns a response event to the core model.

In this approach, core models generate requests to uncore through the L1 cache, which could propagate to all uncore elements looking for the data. Each required cache searches for a hit. The request is forwarded to the next cache level if there is a miss.

This architecture supports multithreaded programs as their workload, which are fully simulated.

3.2 Our Simulation Approach: Simulator Block Diagram

The core models technique splits the simulation of computer architecture functionality into the simulation of core and uncore sub-systems. Both sub-systems interact, so the core emits requests and receives responses from the uncore. Figure 3.2 presents the block diagram of our proposed simulator. It

includes a Trace Decoder, a Core Model Simulator (CMS), an Uncore Simulator, and a Statistics Counter block.

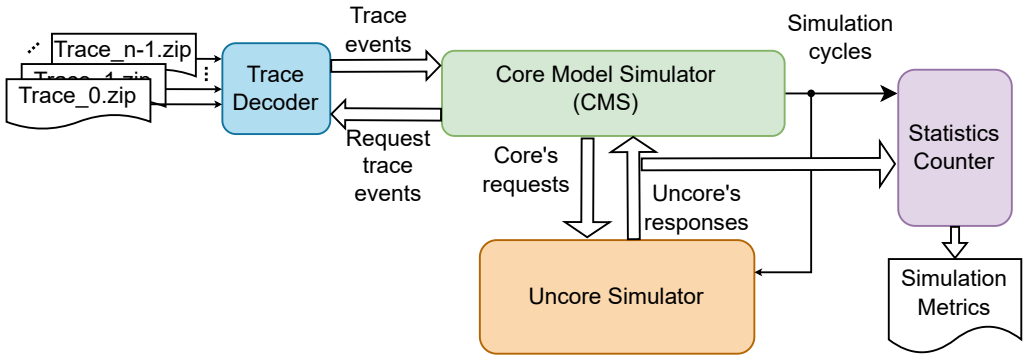


FIGURE 3.2: Block diagram of computer architecture simulator

The Trace Decoder reads n recorded event traces, decompresses, and provides them to the CMS. The CMS simulates traces through independent core models (Section 3.2.1 and Chapter 4 give details of the core models and our implementation). In this way, each core model processes an event trace. When the event is a memory request (memory read or write), the core model propagates the request to the uncore. Then, the uncore processes the requests and updates the core model when it finishes. With this information, the core model computes the memory event's latency. Depending on the simulated event's latencies, CMS computes the simulation cycles. This value represents the estimated cycles to process a benchmark (traces) in the target architecture at any time.

The Statistics Counter registers all uncore's events, such as the number of access hits and misses in all cache levels and memory. Ultimately, the simulation reports these statistics as the simulation results, including the simulation cycles generated by the CMS. Another metric in the simulation report is the computation time spent on the host to run the entire simulation (simulation time). The simulator launcher measures this value.

Section 3.2.1 explains the functionality of the CMS, and Section 3.1 describes the functionality of the Uncore Simulator.

3.2.1 Core Model Simulator

A core model consumes the events, simulates their behavior, and computes their latencies. The simulator computes the latency of one event to know

when the next event in the trace will be processed. The latency of an event depends on its type and the current state of the target architecture.

In our simulator model, to simulate n traces, we instance n core models. This feature potentializes the simulation parallelism because it avoids sequential accesses for two or more traces by a single core (Section 3.3 describes our decision of a parallel implementation of this core model simulator).

Figure 3.3 shows the block diagram of the core model simulator. This diagram illustrates Trace Event Queues (TEQ), a Core Set (CS), and a Timing Controller block.

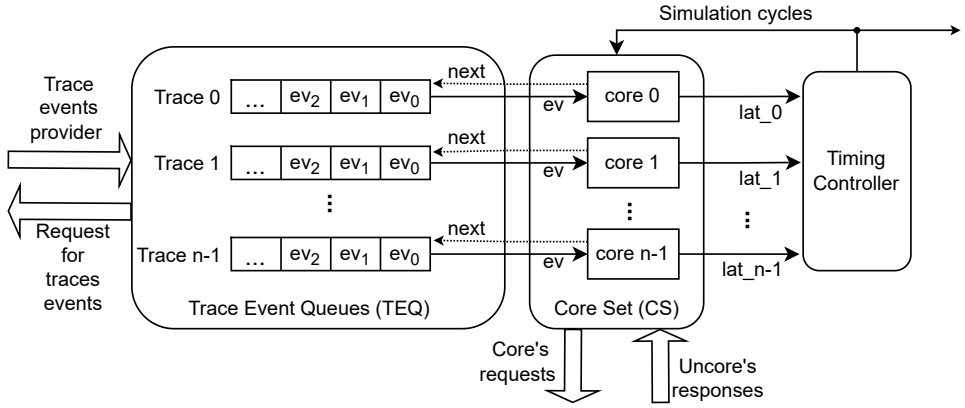


FIGURE 3.3: Core model simulation

We implement limited-size queues of trace events that feed the core models. For each trace, the TEQ implements an event queue. The queue top event is supplied to a core model and it is retired when the core model processes it and requests the next event. Because of the limited-size queue, the request for trace events signal communicates to the trace decoder for new trace events.

During the first simulator's iteration, each core model in the CS block takes the top event(ev_0) from its associated queue, processes it, and communicates the computed event's latencies to the timing controller. If a core model is processing a memory trace event, this event's latency depends on the uncore. Then, the core model generates a request and waits for the uncore's response. When all core models in CS compute their event's latencies, the timing controller finds the minimum latency and sets the next simulation cycle according to this value. Each core model in the second simulator iteration evaluates if the simulation cycle exceeds its simulating event's latency. The core model that meets this condition requests a new trace event to the

TEQ, simulates its behavior, and computes its latency similar to the first iteration. The simulator repeats this process until all events in all traces are processed. A special case in multithreading workloads is the events that depend on other events in other traces. This case will be explained in detail in Chapter 4.

3.3 Tasks Distribution between CPU and GPU

Our objective is to utilize the significant processing power of GPUs for simulating the uncore. An essential aspect to consider when integrating the CMS with the uncore simulator (See Figure 3.2) is whether to implement the CMS as a sequential version on the CPU or a parallel version on the GPU, similar to the uncore simulator.

A sequential version of the core model simulator that can run on a CPU is available since the existing core models have been developed as sequential models. However, in each simulator iteration (refer to section 3.2.1), continuous communication between the core model and uncore is critical, and this communication will occur through PCI. Implementing the core model simulator on the GPU reduces the communication latency, as the uncore and the CMS interact through the GPU main memory. While the GPU main memory typically has tens to hundreds of clock cycles latency [76] [22], a PCI communication represents a latency of tens of thousands of clock cycles [95] [56] [92]. Despite the complexity of parallelizing a system like the CMS, latency is a significant factor, so we decided to implement it on the GPU. Furthermore, this decision allows for exploring a parallelization strategy for the core model.

Then, from Figure 3.2, the *CMS*, the Uncore Simulator, and the Statistics Counter are implemented on the GPU. These components are highly coupled, and they share information (requests, responses, latencies, and the simulation cycle value) through the GPU memory system. Due to the enormous size of traces (from 1 GB to 80 GBs) and limited GPU main memory, the simulator must implement CPU-GPU transfer for trace chunks during the simulation. Additionally, traces are typically in a compressed format. The trace decoder block, running in the CPU, decompresses the traces and arranges the events in a GPU format to be used by the CMS directly (See Figure 3.3). This transfer process requires implementing mechanisms for controlling and

monitoring the trace queues in the TEQ block. Hence, we implemented different GPU kernels to verify the simulation status and control to get new chunks of traces.

Figure 3.4 presents the distribution of the simulator's process on computational resources. This figure shows the host simulation processes and their interaction among them.

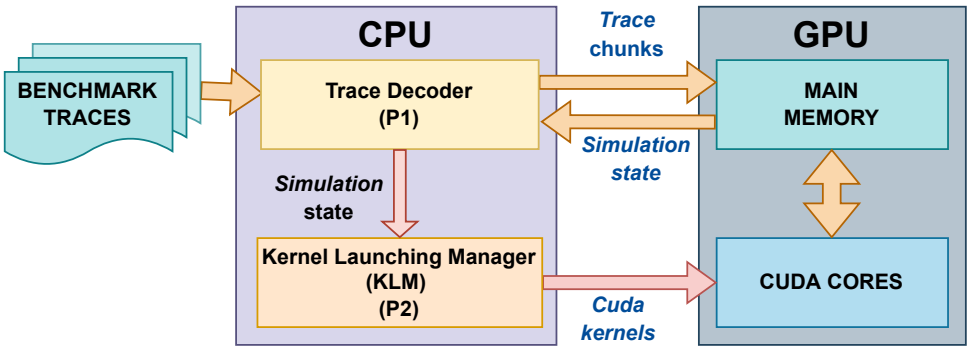


FIGURE 3.4: CPU-GPU process interaction

The *Trace Decoder* process reads the traces from the hard disk. It initializes event structures of these traces in the CPU RAM to transfer them to the GPU main memory when simulation requires it. This process can access the simulation status to know when to transfer new events to GPU. We divided traces into frames of events. Each frame is divided into two zones to serve as a double buffer, in this way, we can separate the transfer data from CUDA kernel processing to hide the memory transfer latencies. A Finite State Machine (FSM) controls the trace events CPU-GPU transfer into the corresponding frame. Each trace has an FSM monitoring and controlling its state during simulation. Figure 3.5 shows this FSM.

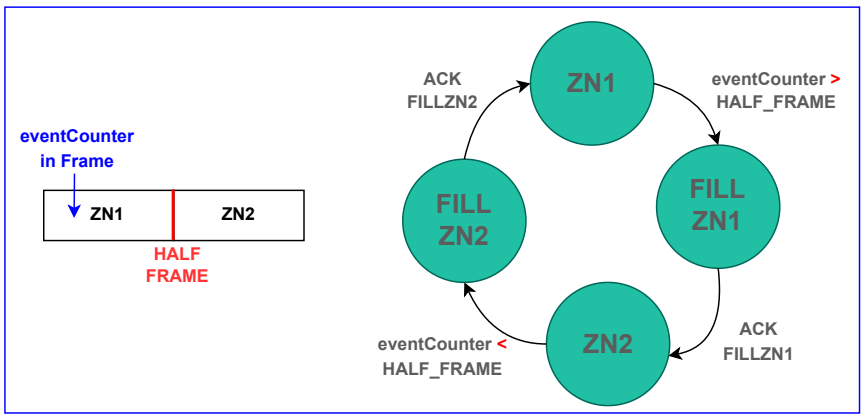


FIGURE 3.5: FSM to control trace events CPU-GPU transfer

The simulation starts processing *ZN1* events. When the simulation processes all zone events in *ZN1*, the corresponding FSM detects this condition and makes a *FILLZN1* request to the CPU; meanwhile, processing continues on *ZN2*. Subsequently, when the simulation processes all *ZN2* events and passes to process events in *ZN1*, the FSM makes a *FILLZN2* request into the CPU. The *Trace Decoder* process continually checks these GPU requests for all traces. If a *FILL* request arrives, this process transfers trace events into the requested zone in the GPU memory and sends an acknowledgment (*ACKFILLZN1* or *ACKFILLZN2*) that the FSM uses. Subsequently, this process reads new events from the hard disc and updates the RAM buffers to the following requests from the GPU.

The Kernel Launching Manager (KLM) process controls the launching of all CUDA kernels for each simulator iteration, for both core model and uncore (See Section 3.2.1 and 3.1).

3.4 Optimizing GPU Processes Runtime

Due to their SIMD architecture and focus on computational units over control flow handling, GPUs are best utilized by programs with low control flow. This approach helps prevent divergence (computational threads follow diverging control paths), which can significantly affect overall performance [27].

Our strategy is to group computer architecture components with similar or identical functionality and model their behavior with the same kernel. For example, we create a GPU kernel to execute the functionality of all core models, and they can execute their function in a parallel fashion. And the same occurs for L1 and L2 caches (See Figure 3.1). During the simulation of a benchmark of n traces in a target architecture of n cores, the simulator has an inherent parallelism for the n cores processing each trace concurrently. Private caches have the same parallelism level. However, when a core simulates a memory trace event, this simulation consists of sequential processes. Figure 3.6 depicts the steps to simulate a particular case (worst-case) when the memory trace event needs data that there is in the main memory. This is a general representation of a computer simulator; chapters 4 and 5 show details of our implementation of this strategy.

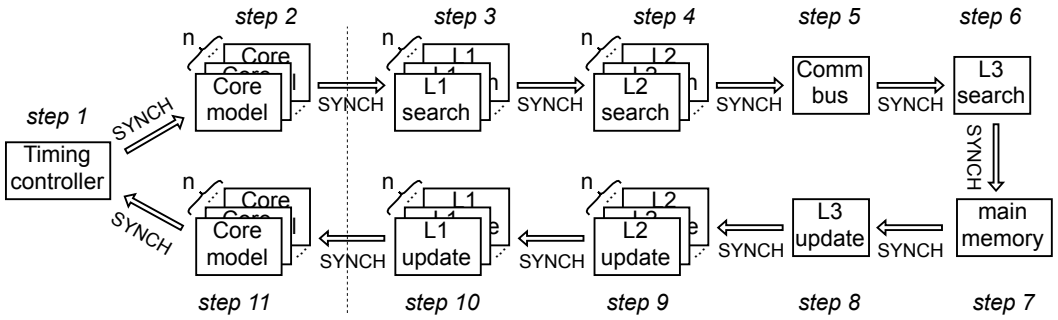


FIGURE 3.6: Sequential steps in a computer architecture simulator

In step 1, the timing controller establishes the simulation cycle value. With this value, all core models process the top trace events (step 2). In step 3, all L1 caches search existing memory requests from the earlier step, and all L2 caches search possible L1 misses in step 4. The communication bus (step 5) takes a request from all existing L2 misses and passes it to the L3 cache (step 6), and last, the request arrives at the main memory (step 7). The following stages update cache memories and give a response to the requesting core model (steps 8 to 10). In step 11, core models compute their event's latencies, and the timing controller can compute the simulation cycles to the next simulator iteration (step 1). The simulator must run a global synchronization (SYNCH) of all existing GPU processes between iteration steps. This synchronization guarantees that all simulation processes have ended and updates the data before the next simulation stage starts.

This sequential model can simulate an event trace ranging from 2 to 11 steps, which depends on the event and the location of the requested data. Moreover, each simulator iteration would delay the event by more steps. A pipeline model is more appropriate to improve the simulator's performance and make it more realistic. With a pipeline model, each simulator iteration would always have two steps. Although this would require more complex control, we implemented this version.

Figure 3.7 shows the 2-step flow graph of our pipelined processes in a simulator iteration. The first step in this graph corresponds to the *timing controller*. The second step contains the kernels representing the target architecture modules, including the core model and the uncore structure processes. Under this approach, while the core model processes trace events, L1 search processes memory requests from the previous simulator iteration, L2 search

processes L1 misses from the last simulator iteration, and so on with the other components.

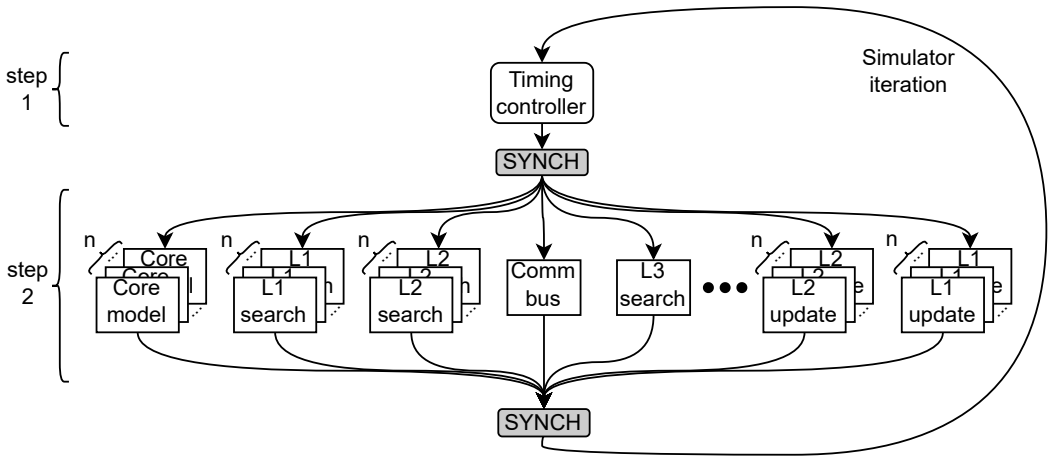


FIGURE 3.7: Pipelined steps in our computer architecture simulator

We implement a double buffer between adjacent kernels to allow event transfer continuously in each simulator iteration and avoid message loss. In a single buffer arrangement, a component that sends two messages in consecutive iterations can rewrite the first message without the receiver component having read the message. Figure 3.8 shows our dual buffer. In the first iteration, all modules write on channel 0 and read channel 1. Then in the subsequent iteration, modules read transferred data in the previous iteration through channel 0. And then, the writing requests use channel 1. This channel switching occurs during the entire simulation.

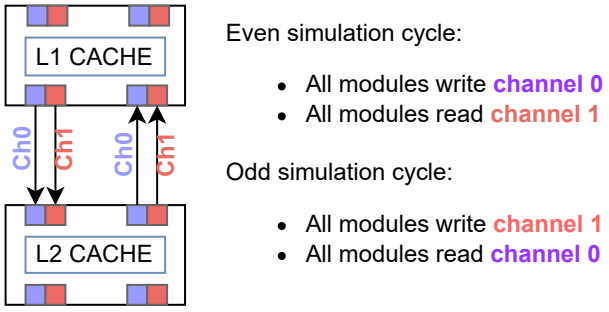
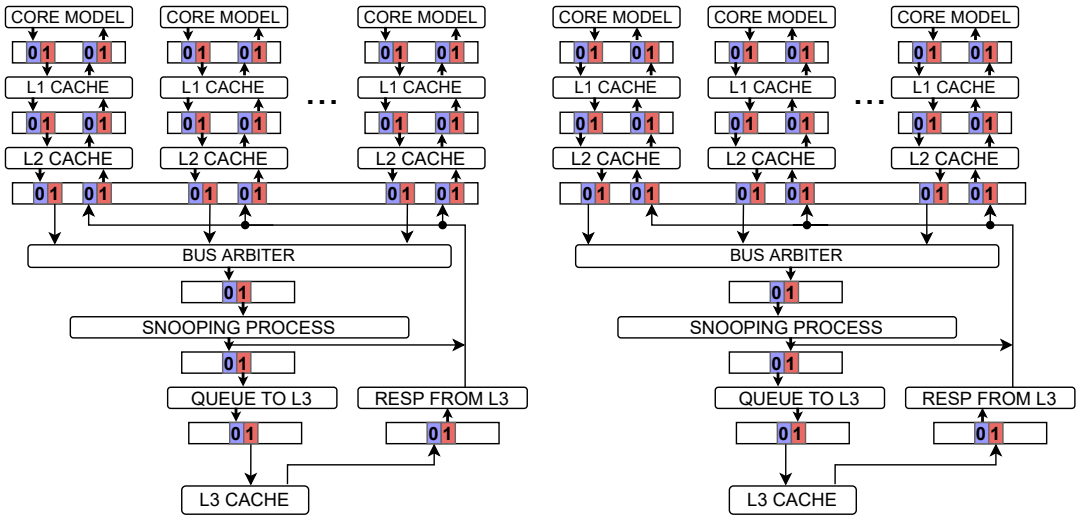


FIGURE 3.8: Double buffer arrangement

Figure 3.9 illustrates the switching buffer for all target architectures during even and odd simulation iterations. During an even simulation cycle, all processes in Figure 3.9a can send messages using channel 0 and check channel 1 for incoming messages. Conversely, during an odd simulation cycle, all

processes can use channel 1 to send messages and check channel 0 for the incoming messages, as depicted in Figure 3.9b.



(A) Message flow in even simulation cycles (B) Message flow in odd simulation cycles

FIGURE 3.9: Message flow in every simulation cycle

The simulation process must launch the simulator iteration millions of times to process all event traces of the benchmark in the target architecture. For this reason, the kernel launch overhead is essential for the simulator's performance. To reduce this overhead, we used the CUDA graph tool [76] to optimize the launching kernel at the level of CUDA threads. From the dependencies flow in Figure 3.7, we can build the CUDA graph assigning each concurrent process to a different CUDA stream. A CUDA stream represents an independent queue of tasks, and it is possible to create dependencies among several CUDA streams. Figure 3.10 shows the CUDA graph that represents our simulation model.

The simulator creates this graph only once and can use it multiple times through a single CPU instruction. The GPU scheduler optimizes launching these graphs and their dependencies [76].

3.5 Summary

In this chapter, we have introduced a simulation strategy that integrates detailed uncore and core model simulations. We've discussed their structures, components, and functionalities, as well as their task distribution and interaction mechanisms. Our proposal involves implementing the core model on

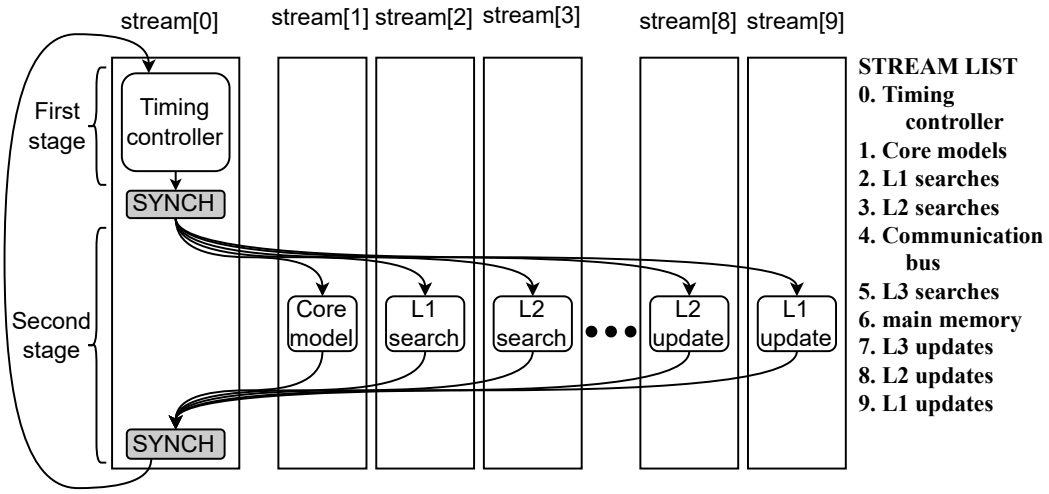


FIGURE 3.10: Cuda graph for our simulation model

the GPU like the uncore, taking into account the communication latency between the CPU and GPU. The primary objective is to leverage the parallel processing power of the GPU. Our approach involves grouping computer architecture components with similar functions and modeling their behavior using common kernels. Additionally, we've outlined the pipeline strategy as a means to enhance the performance of our simulation model, highlighting the essential tools we've developed to boost our model's performance.

Chapter 4

Implementing a Core Model into the GPU

This chapter provides a more in-depth examination of core models, explains the process of selecting a model for our simulator, and describes how it was implemented on a GPU.

4.1 Trace-Driven Core Models

We found various core model flavors that provide favorable balances between simulation speed and accuracy within the single-thread paradigm, such as In-N-Out [54], BADCO [104], and Elastic Traces [41]. These models have reported speedups ranging from 8× to 115× and simulation cycle errors between 5% and 17% compared to detailed simulators. However, since most processors now have multiple cores and can run multithreading applications, there are significant limitations with these core models in supporting these types of architectures.

Sniper [15] is a multithreaded simulator that utilizes the Pinplay framework [81], a dynamic binary instrumentation tool that records the execution of an application into traces. The authors of Sniper have compared it with real machines and found that it can achieve a simulation speed of up to 2 Million Instructions Per Second (MIPS). However, replaying these traces omits the inherent non-determinism in executing multithreaded applications, leading to inaccuracies in design space exploration using simulations of multithreaded workloads [73]. Other approaches, such as Synchrotrace [89] [73], ElasticSimMATE [74], and HETSIM [79], aim to model the non-determinism of multithreaded applications to improve simulation accuracy.

TABLE 4.1: Comparison of Trace-driven models

TRACE-DRIVEN MODELS	Reported SpeedUp Avg.	Reported Simulation Cycles Error	Multi-Threading support	Workloads	Open Source
In-N-Out [54]	115×	7%	N.S.	SPEC CPU2000	No
BADCO [104]	68×	5%	N.S.	SPEC CPU2000, SPEC CPU2006	No
Elastic Trace [41]	8×	17%	N.S.	SPEC CPU2006, HPC	Yes
Sniper [15]	N.A.	25%	PThreads	SPLASH2, PARSEC2, Rodinia	Yes
HETSIM [79]	5×	15.1%	HETSIM primitives	GeMM, GeMV, SpMM	Yes
ElasticSimMATE [74]	3.8×	7.1%	OpenMp, POSIX	MatrixMult, 16x16 to 128x128. HotSpot, k-means, BlackScholes	No
Synchrotrace [89]	9.6×	5.7%	Pthreads, OpenMp	Splash2x, Parsec3, CoMD, Graph500, LULESH, XSBench	Yes

N.A. Not Applicable.
N.S. Not Supported.

Table 4.1 summarizes some of the key features of these core models, including speedup, simulation cycle error, multithreading support, open-source availability, and workload support. However, it should be noted that the values reported in the table do not have standard references, making it difficult to make quantitative comparisons. As a result, the selection criteria for this work were based on qualitative features, such as open-source availability and multithreading support, and the workload benchmarks. In line with our objectives, we placed higher importance on these features and used them to guide our selection.

Out of all the core models on the list, only Sniper and Synchrotrace meet the essential features. However, while Synchrotrace has ample documentation on its model implementation, allowing for replication and modification, the documentation for Sniper is limited. This Sniper limitation makes adapting its model on a parallel platform challenging. Therefore, we have chosen the Synchrotrace model as our base approach as it is publicly available [103], supports workloads written using threading libraries such as Pthreads and OpenMP, and its model is manageable and feasible to parallelize. Although Synchrotrace does not model an out-of-order processor, we have prioritized its other convenient features over this one. Furthermore, it reports an average

speedup of 9.6× and a simulation cycle error of 5.7% compared to cycle-accurate simulation.

4.2 Synchrotrace Core Model

This section provides an overview of the trace generation and trace features in the Synchrotrace core model. The following section also delves into the details of the simulation model in Synchrotrace.

4.2.1 Trace Generation

The Synchrotrace model generates synchronization and dependency-aware traces, one per workload thread. These traces consist of ordered computation, communication, and synchronization events. Synchrotrace utilizes the Valgrind [72] instrumentation framework to generate the traces.

Computation events model the processor execution between the communication or synchronization events. These events contain the number of Integer Operations (IOPs) and Floating Operations (FLOPs) in the core and have the information of memory accesses (reads and writes) in order. Communication events model the data exchange between threads, establishing a read-after-write dependency. A consumer thread can only read data if the producer has written it. Finally, synchronization events model the actions to guarantee synchronism among threads. Synchrotrace supports synchronization mechanisms used by programming models such as Pthreads or OpenMP [18]. It includes seven synchronization events: thread-creation, thread-join, mutex-lock, mutex-unlock, barriers, waiting for conditional variables, and broadcasting and signaling conditional variables. Figure 4.1 summarizes the model's three classes of events.

The Synchrotrace model utilizes an array and matrix to manage mutexes and conditional variables, as Figures 4.2a and 4.2b show. The mutex array contains the state of each mutex (locked or unlocked), which can be accessed by all cores and updated based on a lock or unlock instruction. All cores can access this array, verify, and set the mutex state according to unlock or lock instructions. The conditional variable matrix, used to track which benchmark threads can access a specific conditional variable, is updated when a

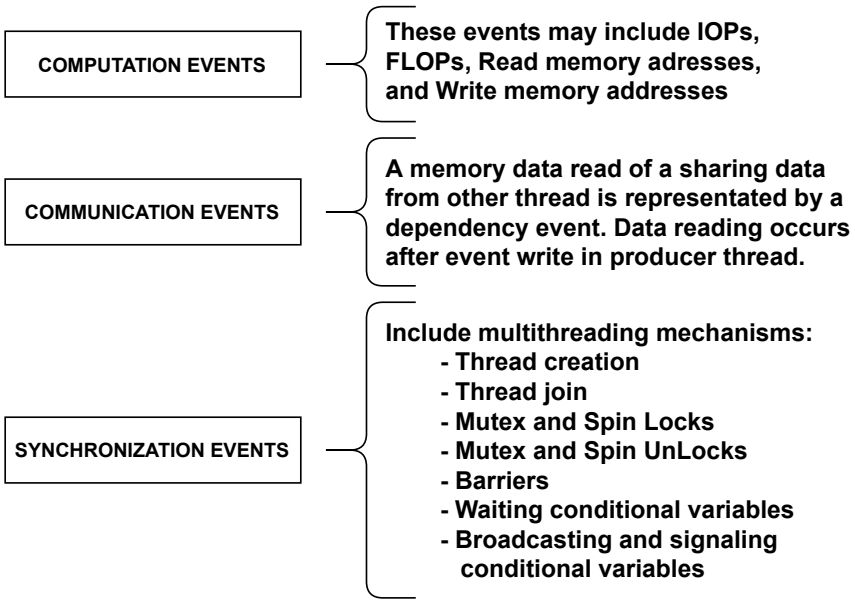


FIGURE 4.1: Synchrotrace events types

signaling or broadcasting event occurs. The simulator sets an entire row for the conditional variable when a broadcasting event occurs.

$$\begin{array}{cccccc} Mtx_0 & Mtx_1 & Mtx_2 & ... & Mtx_{m-1} \\ (\ 0 & 1 & 0 & ... & 1\ ) \end{array}$$

(A) Mutex array

$$\begin{array}{c} \\ CV_0 \\ CV_1 \\ CV_2 \\ ... \\ CV_m - 1 \end{array} \begin{array}{c} \begin{array}{ccccc} Th_0 & Th_1 & Th_2 & ... & Th_n - 1 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & ... & 1 \\ 1 & 0 & 0 & ... & 0 \\ 0 & 1 & 0 & ... & 1 \\ 0 & 0 & 1 & ... & 1 \\ 0 & 0 & 0 & ... & 0 \end{pmatrix} \end{array}$$

(B) Conditional variable matrix

FIGURE 4.2: Mutexes and conditional variables arrays

4.2.2 Synchrotrace Simulation

In the trace simulation or playback stage, the trace-event simulator processes the header events for each trace (queue) and retires the event when the event's latency is known to give way to the next queued event. Synchrotrace reads

the events in the traces sequentially and generates the needed actions to playback according to the simulating event type. The simulator must evaluate the new header events in the next simulator iteration.

The playback stage implements three processes to simulate the events, the Event Queue Manager (EQM), the Memory Request Manager (MRM), and the Thread Synchronizer (TS). Figure 4.3 visually represents these processes and their interactions [89]. The MRM handles memory requests and interactions with the uncore model, generates read and write requests, and manages responses.

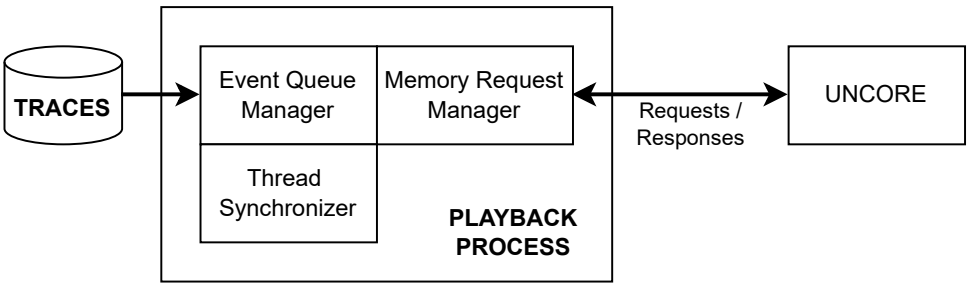


FIGURE 4.3: Synchrotrace playback process

The EQM is the backbone of the Synchrotrace simulator. This component examines the availability of trace events to simulate in each iteration of the simulator. It also monitors the dependencies between trace events, resolves their progression, and manages the scheduling of upcoming events to simulate. The EQM handles each event type and removes them from the queues, as outlined in Algorithm 4.1 [89].

Computation events schedule their memory requests after the computation time has elapsed. If the memory request is of type *READ*, the event is retired once the uncore responds through the MRM block, as outlined in lines 3 to 8 of the algorithm. When a dependency in a communication event is fulfilled, the read request pass to the MRM, and the event waits for the memory response as per lines 9 to 14. Lines 15 to 36 detail the processing of each subtype of synchronization events. The create and join events are passed to the TS to manage a thread's creation or destruction (joining). A barrier event waits for all threads that share the same barrier. Mutex unlock enables a mutex, and a conditional variable broadcasting enables the variable for all other threads, as outlined in Figures 4.2a and 4.2b. Mutex lock and conditional variable wait events are retired if the mutex or the conditional variable is enabled.

Algorithm 4.1 Algorithm for Synchrotrace ProcessEvent Function

Input: $eventToProcess$
Output: $None$

```
1: function SynchrotraceProcessEvent()
2:     switch event.type do
3:         case COMPUTATION
4:             IssueMemReq@(CurrentTime+compTime)
5:             if request.type == READ then
6:                 WaitForResponse
7:             else
8:                 QueueNextEvent
9:         case COMMUNICATION
10:            if DependentEventCompleted then
11:                IssueMemReq@(CurrentTime)
12:                WaitForResponse
13:            else
14:                ScheduleThreadNextCycle
15:        case SYNCHRONIZATION
16:            if synch.type == CREATE or JOIN then
17:                SendEventToThreadScheduler
18:            else if synch.type == BARRIER then
19:                if LastThreadForBarrier then
20:                    QueueNextEvent
21:                else
22:                    ScheduleThreadNextCycle
23:            else if synch.type == MUTEXLOCK then
24:                if MutexLockObtained then
25:                    QueueNextEvent
26:                else
27:                    ScheduleThreadNextCycle
28:            else if synch.type == MUTEXUNLOCK then
29:                QueueNextEvent
30:            else if synch.type == CONDVBLE_WAIT then
31:                if CondVble == 1 then
32:                    QueueNextEvent
33:                else
34:                    ScheduleThreadNextCycle
35:            else if synch.type == CONDVBLE_BROADCASTING then
36:                QueueNextEvent
```

4.3 Modifying Synchrotrace Model

The simulation process outlined in Section 4.2.2, while efficient for execution on a CPU, has a complex control flow that needs better-suited execution on a GPU. Our strategy involves creating specialized kernels to reduce control divergence for addressing its GPU implementation. We split the processes into kernels based on event type (computation, communication, and synchronization), and each kernel only simulates events of its corresponding type. However, these subprocesses still contain control flow divergences

and joint operations. To further improve efficiency, we transform the original Synchrotrace events into simple ones that reduce the control flow and can be processed more uniformly. This event transformation allows the proposed model to simulate six original event types using only three simple new ones. Figure 4.4 illustrates the relationship between the original Synchrotrace events on the left and the proposed ones on the right.

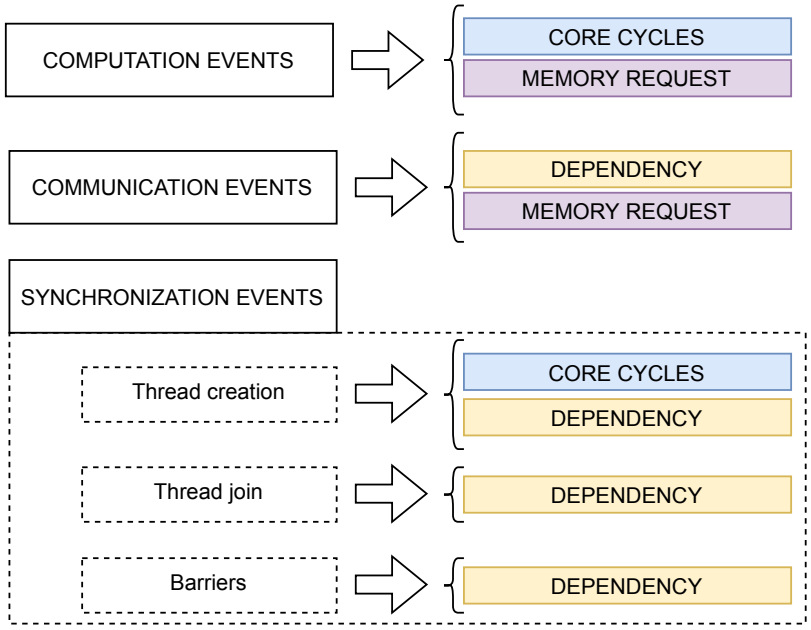


FIGURE 4.4: Breaking down Synchrotrace events

The transformation splits computation events into two distinct types: *core cycles* and *memory request* events. *Core cycle* events represent the time a core executes instructions and include the number of integer and floating operations. The simulated core's abstraction level determines the number of cycles for each operation type. On the other hand, *memory request* events handle requests to the uncore and wait for a response. These events include the type of access (read or write) and the target memory address.

The transformation process also separates communication events into *dependency* and *memory request* events. A *dependency* event depends on a specific trace event and blocks the execution of the trace until the independent event is completed (as explained in Section 4.2). Thread creation events are divided into *core cycles* and *dependency* events. The thread creation event includes a *core cycles* event in the creator thread and a *dependency* event at the beginning of the created thread, waiting for an event in the creator thread. Lastly, *thread join* and barrier events are transformed into *dependency* events. A *thread join*

event is a *dependency* event that waits for the last event in the joined thread, and a barrier event is a list of *dependency* events that wait for each other's threads to reach the barrier point. As a result of this transformation, the new model eliminates the need for the thread scheduler block of the original Synchrotrace version (as shown in Figure 4.3) and any additional data structures to manage threads reaching the barriers.

The transformation process preserves the structure and information of *mutex lock*, *unlock*, *waiting*, and *broadcasting* of *conditional variable* synchronization events. New traces have more events than initial traces, but the events in the new traces are generally simpler and have less information. This simplification makes it easier and more practical to parallelize the simulation of the transformed core model compared to the original version.

We established three potential states (*READY_TO_RUN*, *WAITING*, or *COMMITTED*) and three potential actions for simulating the new events. *READY_TO_RUN* is an initial or default state; all events in the queue top will have this state. In a simulator iteration, if the event with this state fulfills all conditions to be retired, it will transition to the *COMMITTED* state and be replaced by the next event in the queue. On the other hand, if the event does not fulfill the conditions for retirement, it will transition to the *WAITING* state. It will be re-evaluated in the subsequent simulation iteration.

The simulation process outlined in Algorithm 4.2 evaluates and processes events based on their state. The core model simulator runs until all traces have completed their events. A nested loop will iterate through the top events of each trace and handle them based on their current state. This nested loop represents one iteration of the simulator, as detailed in Section 3.2.1.

Algorithm 4.2 Event state evaluation

Input: $eventQueues$
Output: $currentExecTime$

```
1: while ∃ events in queue do
2:     for ∀ traces do
3:         if eventState == READY_TO_RUN then
4:             startTraceEvent()                ▷ eventState= COMMITED or WAITING
5:         else if eventState == WAITING then
6:             waitConditions()                 ▷ eventState= WAITING or COMMITED
7:         if eventState == COMMITED then
8:             LaunchTime[thisTrace] = currentExecTime+EventLatency
9:     currentExecTime = min(LaunchTime[∀ traces])
```

The *startTraceEvent()* function, as described in Algorithm 4.3, evaluates events with *READY_TO_RUN* state. The *waitConditions()* function, as outlined in Algorithm 4.4, checks for conditions to be met. When an event meets the conditions to finish, it sets its state to *COMMITTED* and computes its latency. The event is then retired from the simulation queue, and the simulation determines the next event's launch time by adding the committed event's latency to the current simulation cycle. The simulation updates the execution time using the minimum value among launch times of all top events when the *for loop*, which iterates the top events in each trace queue, finishes (See line 9 in Algorithm 4.2). This current execution time will serve as a reference for the next simulation cycle. At this point, a new simulator iteration starts again, and the process repeats until there are no more events to simulate.

Algorithm 4.3 outlines the *startTraceEvent()* function, which handles events in the *READY_TO_RUN* state as specified in Algorithm 4.2. The function calculates the latency of the event and sets its state based on the event type. For *CORE_CYCLES* type events (Lines 3 to 5), the *startTraceEvent()* computes the latency as the product between integer operations (IOPS) and the cycles per integer (CPI) operation plus the multiplication between floating operations (FLOPS) and cycles per float (CPF) operation. Then, the function sets the event state to *COMMITTED*. The function also directly processes events of types mutex unlock (*MUTEX_UNLOCK*) and broadcasting of a conditional variable (*CONDVBLE_SIGNAL_BROADCASTING*) (Lines 6 to 9), setting their latency to 1 and their state to *COMMITTED*.

When the events are *DEPENDENCY*, *MUTEX_LOCK*, or conditional variable wait (*CONDVBLE_WAIT*) types (Lines 10 to 18), the function checks if the associated condition has been met and sets the event latency to 1, otherwise, the event latency will have an infinity value, and the event state will be *WAITING*. An event in the *WAITING* state will be evaluated in the next simulator iteration by the *waitConditions()* function and will do so until the event fulfills the condition. This function processes the events of type *MEMORY_REQUEST* (Lines 19 to 22) by sending requests to the uncore and setting the event's latency to infinity and its state to *WAITING*. Like *DEPENDENCY* events, *MEMORY_REQUEST* events will be processed in the following simulator iterations by the *waitConditions()* function until the uncore notifies it that the request was processed, and the simulator knows which is the event's latency.

Algorithm 4.3 Algorithm for the **startTraceEvent** Function

Input: $eventType$
Output: $eventState$, $eventLatency$

```
1:  function STARTTRACEEVENT()
2:      switch eventType do
3:          case CORE_CYCLES
4:              eventLatency = IOPS*CPI+FLOPS*CPF
5:              eventState = COMMITTED
6:          case MUTEX_UNLOCK
7:          case CONDVBLE_SIGNAL_BROADCASTING
8:              eventLatency = 1
9:              eventState = COMMITTED
10:         case DEPENDENCY
11:         case MUTEX_LOCK
12:         case CONDVBLE_WAIT
13:             if condition_OK then
14:                 eventLatency = 1
15:                 eventState = COMMITTED
16:             else
17:                 eventLatency = ∞
18:                 eventState = WAITING
19:         case MEMORY_REQUEST
20:             requestToUncore()
21:             eventLatency = ∞
22:             eventState = WAITING
```

The *waitConditions()* function, as outlined in Algorithm 4.4, checks the conditions of events in a *WAITING* state. Specifically, it checks if the uncore has fulfilled *MEMORY_REQUEST* events or if *DEPENDENCY*, *CONDVBLE_WAIT*, or *MUTEX_LOCK* events have met their respective conditions.

Algorithm 4.4 Algorithm for the **waitConditions** Function

Input: $eventType$
Output: $eventState$, $eventLatency$

```
1:  function WAITCONDITIONS()
2:      switch eventType do
3:          case DEPENDENCY
4:          case CONDVBLE_WAIT
5:          case MUTEX_LOCK
6:              if condition_OK then
7:                  eventLatency = 1
8:                  eventState = COMMITTED
9:          case MEMORY_REQUEST
10:             if responseFromUncore then
11:                 eventLatency = eventLatencyFromUncore
12:                 eventState = COMMITTED
```

4.4 Implementing a Parallelized Version of the Core Model

This section explains how the Synchrotrace core model was parallelized for its execution in GPU. The details of the trace data structure used in the GPU-based simulator are outlined, and the methods for running core model processes concurrently are discussed. To optimize the simulation of Synchrotrace traces, we leverage the data parallelism capabilities of the GPU by assigning each trace to a CUDA thread. This approach allows us to simultaneously execute the event-related kernels using CUDA streams, resulting in increased parallelism and improved computing efficiency as more threads are utilized.

4.4.1 Description of the Trace Events Data Structure

To optimize the core model simulator for GPU processing, we employed a data structure that ensures uniformity in the storage of events, with each event type occupying the same number of bytes, as depicted in Figure 4.5.

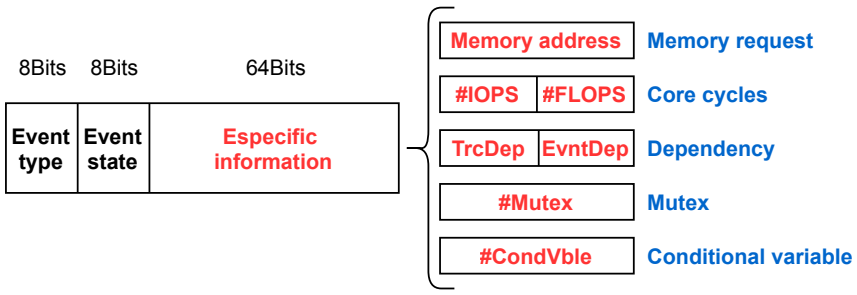


FIGURE 4.5: Regular structure to store GPU-based events

The Event type field serves as an identifier for the types outlined in Section 4.3, and the specific information field depends on the event type's value. For example, if the event is a memory request, this field contains the memory address (64 bits) associated with this request. In the case of a *core cycles* event, this field includes the number of IOPs and FLOPs (32 bits each). *Dependency* events have in this field the trace and event number that causes the dependency (32 bits each). *Mutex* and *conditional variable* events store the identifier for the specific *mutex ID* or *conditional variable* implicated in the event. Lastly, the *Event state* field holds a value evaluated by Algorithm 4.2.

In our simulation model, the *Trace ID* and *Event ID* fields are unnecessary for all events as they can be inferred by the trace's memory layout and the event's position in the sequence, which results in more efficient processing and reduces the need for extra memory storage. To improve memory alignment, we padded [12] the event structure to 128 bits, despite requiring only 80 bits. We prefer unused memory space that overloads the simulation with multiple access to get a misaligned event. A padded structure is also preferred instead of more processes to uncompress the event information for a compressed format for events in fewer bits (64).

4.4.2 Event Processing through GPU Kernels

We convert Synchrotrace events into a uniform format to simplify the simulation process, as outlined in Section 4.3. Consequently, we develop separate kernels to handle each event type and apply the corresponding case of Algorithm 4.3. As shown in Figure 4.6, we have created seven kernels to process each trace event type. The kernel dimensions match the number of traces. The kernels concurrently process the *n*-trace events for each simulator iteration, with each kernel thread checking and simulating if the event matches its corresponding type.

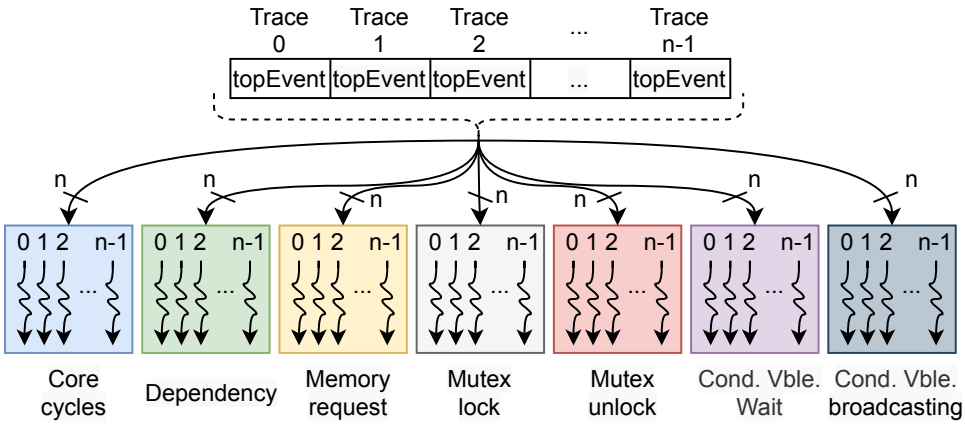


FIGURE 4.6: Data and task parallelism in the core model simulator

Appendix A.1 presents the pseudocode of all core model kernels that run concurrently to process each trace's top event. The kernels receive the trace and event counters as parameters to determine which event to check. If a CUDA thread detects that an event corresponds to its type, it will process it and set the latency values as Algorithm 4.3 describes.

The *core cycles* kernel thread retrieves the integer and floating operation counts (IOPS and FLOPS) from the event structure and calculates the number of core cycles the event uses. Kernel threads calculate these core cycles by multiplying the IOPS and FLOPS by the number of cycles per operation, a user-configurable parameter in the simulator. The kernel thread then uses the core cycle value to determine the launch time for the next event.

The *dependency* kernel ensures the fulfillment of the requirements indicated by the corresponding events. The dependency, which consists of *TrcDep* and *EvntDep*, is stored in the event structure (Figure 4.5). The kernel threads check if the corresponding event has fulfilled the top event's dependency. The simulator removes the event from the trace's top if it has. If the event does not meet its condition, it will be re-evaluated in the next simulation cycle. The *memory request* kernel extracts the address and request type from the event structure, sending the requests to the uncore and updating the event's state to *WAITING* (See Section 4.3). This kernel also checks if the uncore has finished processing the request and updates the launch time accordingly, allowing events with a launch time equal to or less than the current simulation time to be processed.

The *mutexLock* and *mutexUnlock* kernels simulate the effect of software mutex synchronization during trace simulation using a flag array to keep track of the mutex's state (locked or unlocked). Each mutex has an *ID* and a state stored in the flag array (See Figure 4.2a). The CUDA threads of the *mutexLock* kernel check the mutex state in the flag array. If the mutex is unlocked, the flag array updates the mutex's state to locked, and the *mutexLock* event is retired. In contrast, if the mutex's state is locked, the event changes to *WAITING* and is re-evaluated in the next simulator iteration. The CUDA threads of the *mutexUnlock* kernel update the mutex state related to the event. In any case, the event is retired.

The *condVbleWait* kernel emulates a conditional variable's waiting time until another thread sets the variable. The CUDA threads of the *condVbleWait* verify the state of the event-associated variable on the conditional variables matrix to decide on retiring the event or keeping waiting. If the variable state is one, the event is retired, and the variable state is clear. Otherwise, the event remains in the trace for re-evaluation in the next simulation cycle. Figure 4.2b illustrates the conditional variable matrix. Our trace simulation model treats

all individual *signal* events as broadcasting events to avoid potential deadlocking issues. This approach aligns with the original Synchrotrace simulation model, as it also avoids the need to capture the specific predicate that the variable must wait. The *condVbleSignalBroadcasting* kernel identifies the events that set a specific conditional variable. This setting can be done concurrently for the conditional variable in all traces (See Figure 4.2b).

4.4.3 Implementing the Core Model Timing Controller

We use the structure illustrated in Figure 4.7 to control the global simulation cycles and the scheduling of the next iteration in the simulator.

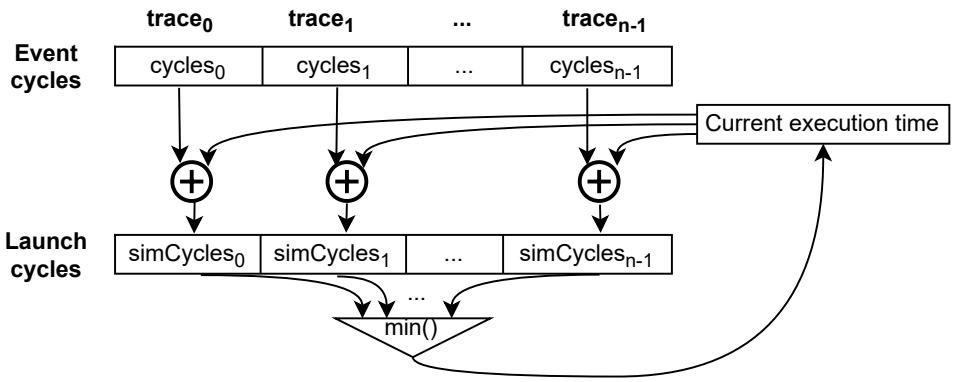


FIGURE 4.7: Simulation timing control

The *launch cycles* array keeps track of when each trace will execute its next event, while the *global simulation cycles* variable holds the current simulation timing value. This value is determined by finding the minimum value in the *launch cycles* array and serves as the reference for evaluating each queue header event in each simulation loop. The *event cycles* array records the cycles (latency) of each event processed in the current simulation loop. To update the *launch cycles* array, we add the *global simulation cycles* with the corresponding event cycles. To efficiently determine the minimum latency (cycles) among all events, we employ a reduction algorithm, which reduces the computational complexity from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$ [108].

4.5 Summary

In this Chapter, our selection process led us to choose the Synchrotrace model as our core model. Given the limitations of the available models, we made

our choice based on qualitative factors, including its support for multithreading workloads and its adaptability for integration into an uncore simulator. Furthermore, according to its report, it is both fast and accurate.

We embarked on a transformation process to make Synchrotrace compatible with our GPU-based uncore simulator. This transformation involved streamlining the traces to contain less information, which greatly facilitated parallelization and improved performance. Additionally, we implemented the core model in a parallel version for the GPU, which encompassed the establishment of a structured event data system, event processing, and the introduction of a timing controller.

Chapter 5

Parallelizing the Uncore Simulator for a GPU

This chapter details the uncore structure and outlines our methods for parallelizing the uncore simulation model for GPU use. We discuss our parallelization techniques for each level of the uncore simulator and provide a performance analysis of the parallelized simulator.

5.1 Uncore Architecture

Section 3.1 presented a basic version of the uncore structure for a multicore architecture, which is the foundation for the complete version shown in Figure 5.1. This version includes a coherence protocol [71] and implements a write-back and Not Recently Used (NRU) replacement policy [42] for all caches. As described in Section 5.1.1.2, the NRU policy is well-suited for parallel implementation. The uncore structure presented in this chapter allows the customization of various parameters, including latencies for each hierarchy level, the latency of the snooping process, cache sizes, associativity for each cache level, and line sizes.

As described in Section 3.1, this architecture includes n cores and the same number of private L1 and L2 caches. An architecture *node* in this multicore structure includes the private elements (core, L1, and L2 cache). The subsequent sections will provide further details on each component of the uncore.

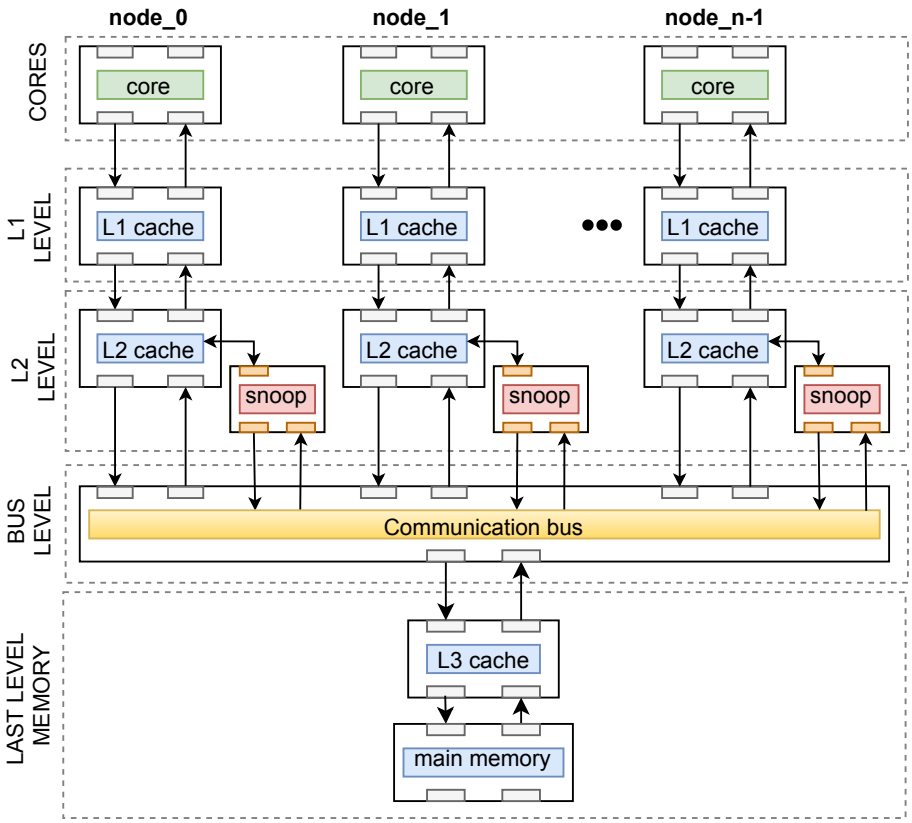


FIGURE 5.1: Target computer architecture with coherence mechanism

5.1.1 Cache Memories

In our model, the basic unit of cache memory is the line or block, and a cache is a collection of these lines. Each line comprises a state field, a marker for the replacement policy, and a tag field. Figure 5.2 illustrates this line structure.

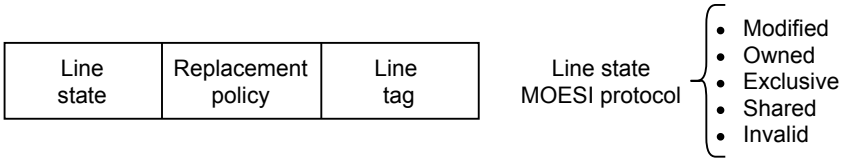


FIGURE 5.2: Fields in a cache line

The coherence protocol in our model assigns different states to the *line state* field, such as *Modified*, *Owned*, *Exclusive*, *Shared*, or *Invalid* (MOESI) [71]. The Not Recently Used (NRU) policy uses binary values, 0 or 1, for the *replacement policy* field.

The number of lines, p, in a cache is determined by dividing the cache size by the line size ($p = CacheSize/LineSize$). Figure 5.3 illustrates the array of lines for a cache and their initial values. The simulator initializes the *line state* fields with an *Invalid* state and the *replacement policy* field with a zero value. The *tag* field is only valid if the *line state* field has a value other than *Invalid*.

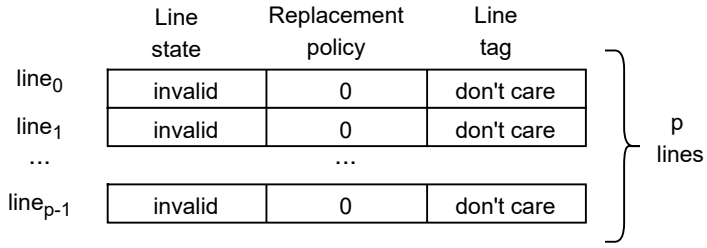


FIGURE 5.3: Array of lines in a cache

N-way set-associative caches create groups of lines called sets. A line can occupy any way of a specific set. Figure 5.4 illustrates the line sets in an 8-way associativity cache. The number of sets in a cache, s, is calculated by dividing the number of lines, p, by the associativity grade, N ($s = p/N$). The value of N represents the associativity grade, with higher values allowing for more efficient use of the cache but also increasing its complexity and cost [71] [40].

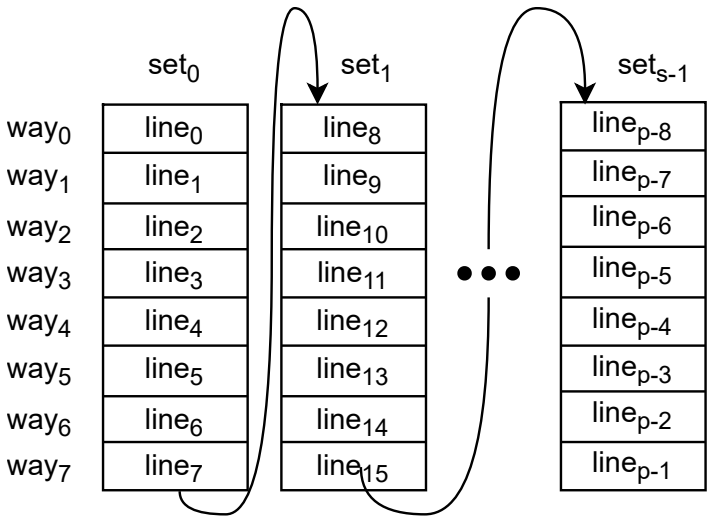


FIGURE 5.4: Line sets in an 8-way associativity cache

We arranged the cache arrays sequentially at a specific level for all nodes. For example, in all L1 caches, we have an array with the p lines of *node_0*, followed by the p lines of *node_1*, and so on, until the p lines of *node_n-1*. Similarly, in the L2 cache, we have another array with the same sequence.

This organization allows us to efficiently process requests to all L1 caches by indexing this ordered array simultaneously (Computational complexity $\mathcal{O}(1)$). Figure 5.5 illustrates this arrangement of cache lines at a specific level.

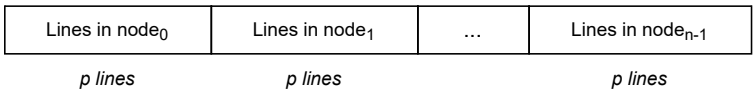


FIGURE 5.5: Array of cache lines at a specific level

In our model, when a request or response arrives from another component, the cache processing mechanism operates on the data structures to search for a hit or to locate a new line. We have implemented non-blocking channels through which messages can be sent and received to facilitate communication between adjacent architectural components. As shown in Figure 5.6, there is one channel for requests and another for responses between the L1 and L2 cache. When a sender component writes a message to the corresponding channel, it remains there until the receiver reads and removes it. Sections 5.1.1.1 and 5.1.1.2 describe the processes and message structures to manage requests and responses in the cache.

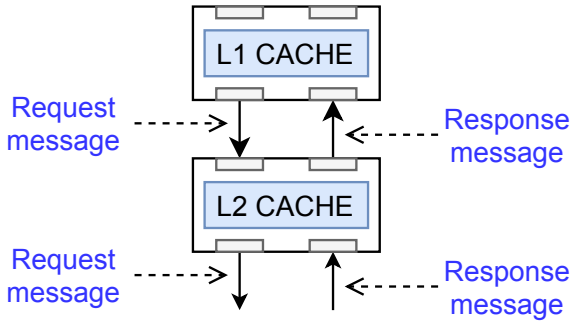


FIGURE 5.6: Interblock communication channels

5.1.1.1 Managing Cache Requests

The structure of the request message is illustrated in Figure 5.7 and includes three fields: the type of request, the data address, and the node that made the request.

Requests for reading and writing memory data come from the core. The outcomes of replacement policies and coherence protocols in caches can include the following:

- a write-back operation,

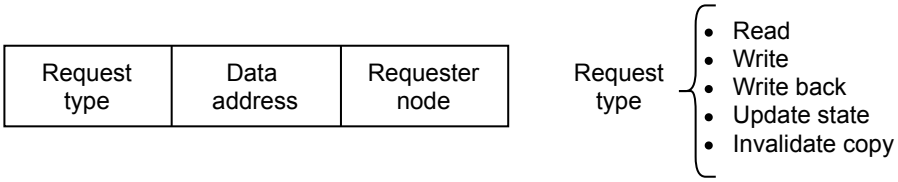


FIGURE 5.7: Request message structure

- a request for a state update at a lower level, or
- a request to invalidate copies in other nodes.

The request message includes the data address involved in the transaction and the identification of the node that made the request. This identification helps determine which node requires the response in communication among shared resources.

When a request is received, the cache searches for a hit in its line array by identifying the corresponding set and checking each way for a valid line state and a matched tag. The process checks the *Line state* and the *Line tag* fields in each way of the searched set to determine the hit condition. Figure 5.8 provides a more detailed view of the structure for searching a hit in a cache set. The validation component's output depends on the request type (read or write) and the coherence protocol implemented. Section 5.1.3 explains the *MOESI* coherence protocol and how the validation component performs its function.

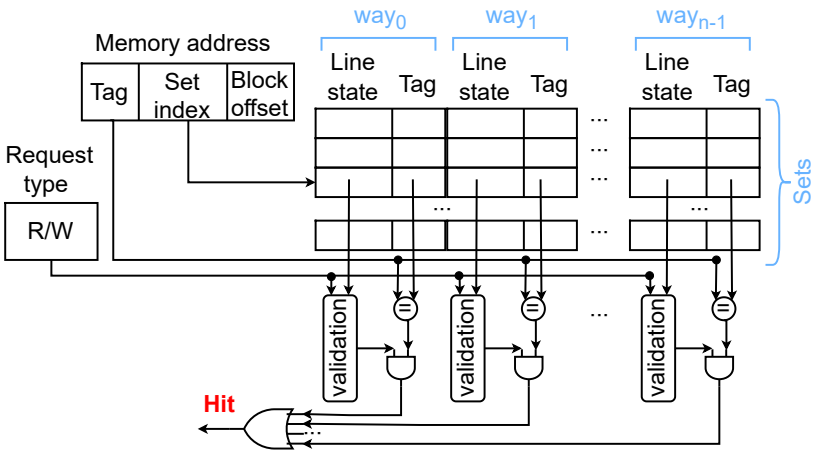


FIGURE 5.8: Detailed description of the process to determine hit/miss

5.1.1.2 Managing Cache Responses

The cache sends a message in response to a request. Figure 5.9 illustrates the cache response's message structure.



FIGURE 5.9: Response message structure

This structure has four fields, *response type*, *data address*, *requester node*, and *data state*. The response type indicates whether the request resulted in a hit or a miss. The *data state* field shows the state of the requested data. In an in-order execution model, missed responses are unnecessary, as the core model will wait for a hit response. Miss responses are forwarded to higher levels in the memory hierarchy but only utilized by the statistic counter block to generate the simulation report (refer to Section 3.2). As a means of prematurely ending its execution in the event of abnormal cases, the simulation will automatically abort if, for any reason, the requester core model does not receive the expected hit response.

When a response arrives at a specific cache from a lower level, a process determines where to place the incoming data. This process involves checking the *Line state* and *Replacement* fields, and the decision depends on their values (as shown in Figure 5.2). The placement checker selects a way to randomly place incoming data from a list of eligible candidates. Any line with an invalid value in its *line state* field is a candidate, but if there are no lines with this state value, any line with the *replacement policy* bit indicating that it has not been used recently will be a candidate. Figure 5.10 illustrates this process. When no lines are eligible for incoming data, all lines in the set are considered candidates. In order to minimize performance impact, our implementation opts for selecting line 0 instead of choosing randomly. If the chosen line is modified (dirty state), the cache sends a write-back request to the next cache level.

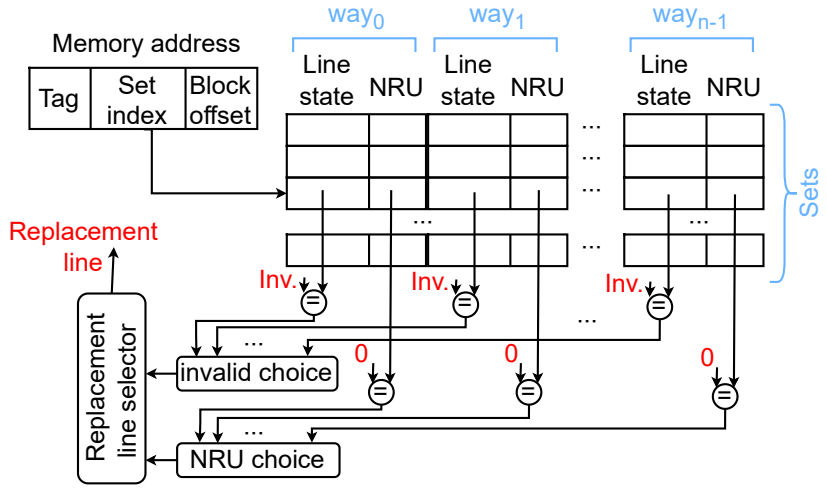


FIGURE 5.10: Detailed description of the placement process

5.1.2 Communication Bus

The shared bus facilitates communication between nodes and the L3 cache. It enables nodes to make data requests and sends responses back to the requesting node. Our uncore model breaks down the bus process into several subprocesses and uses a pipeline approach to handle multiple bus requests simultaneously. Figure 5.11 illustrates these subprocesses.

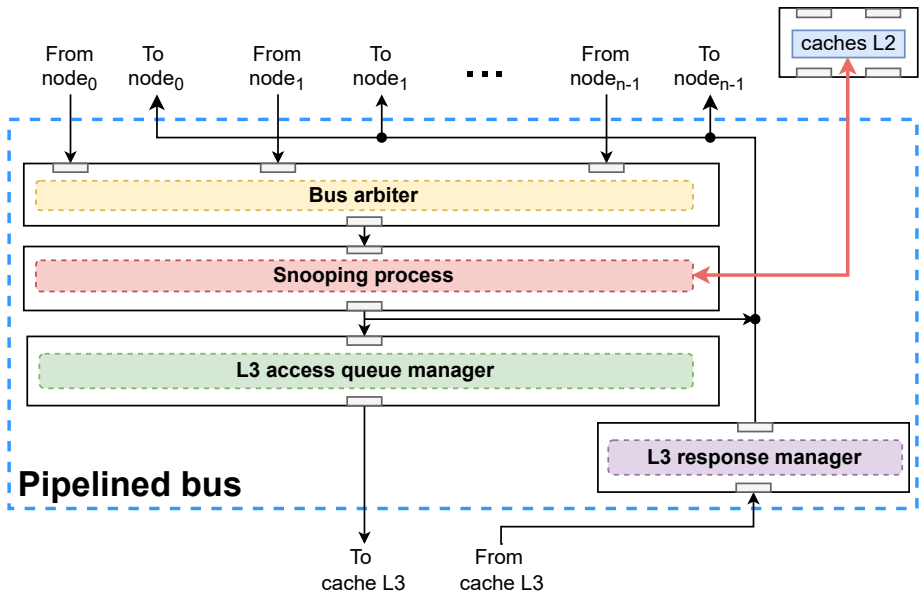


FIGURE 5.11: Communication bus subprocesses

The *bus arbiter* assigns exclusive access to the bus resources to one request at a time by checking L2 ports in all nodes for requests. This assignment can be through a round-robin, priority, or random policy. In particular, we implement a random policy. The arbiter registers the request's line address until

shared resources process that requests, preventing the simultaneous resolution of multiple requests with the same line address.

In the *snooping* stage, the arbiter sends the selected request through the snooping port. The *snooping process* then searches all L2 cache-level nodes to check if the requested line address exists. If the search finds the data, the coherence protocol's updating process runs as described in Section 5.1.3. However, if the search does not find the data, the process generates a miss snooping and forwards the request to the next level, the *L3 access queue manager*.

The access to the L3 cache is handled by the *L3 access queue manager* through a FSM depicted in Figure 5.12. When L3 is idle, and a request from snooping arrives, the *L3 access queue manager* forwards the request to L3, and the FSM transitions to the *BUSY L3* state. If another snooping request arrives while L3 is busy, it is added to a queue. Once L3 completes the task, the *L3 response manager* releases the L3 cache access and sends the response message to the requester node. The FSM then returns to the *IDLE L3* state, ready to handle new snooping requests or requests already in the queue, causing it to transition to the *BUSY L3* state again.

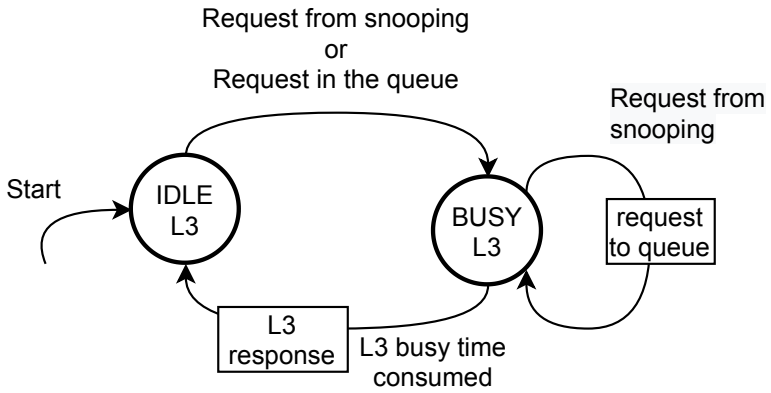


FIGURE 5.12: State machine for L3 access queue manager

5.1.3 MOESI Snoop Process

This section explains the MOESI protocol [71], its modeling, and its implementation. The bus snooping and the cache requesting manager (See Section 5.1.1.1) processes carry out the MOESI protocol. The first time a line is read, the main memory and L3 cache respond, and the requester node will have the line with the *Exclusive* state. If another node reads the same line, the state changes to *Owned*, and the requester node will have the line in a *Shared*

state. A writing request from the core changes the line's state to *Modified*. Figure 5.13 illustrates the meaning of each state in a cache line.

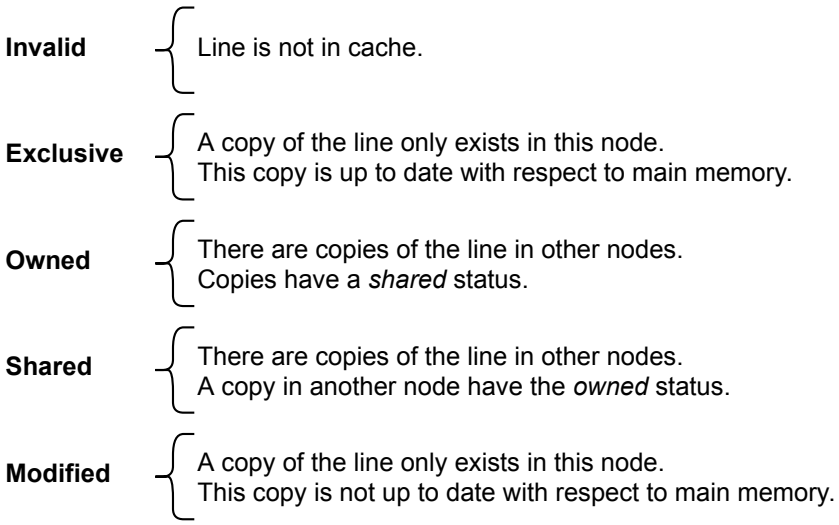


FIGURE 5.13: Meaning of states in the MOESI coherence protocol

The state diagram in Figure 5.14 illustrates the process of the requesting cache. Suppose a request is a read operation, and the line is present in a cache with a state of *Modified*, *Owned*, *Exclusive*, or *Shared*. In that case, the requesting cache process results in a cache hit, and the line state remains unchanged. However, if the line has an *Invalid* state in the private caches, this process results in a cache miss, and the request passes to the next level. If the line is absent in any node, the main memory and L3 cache will respond with the line in the *Exclusive* state.

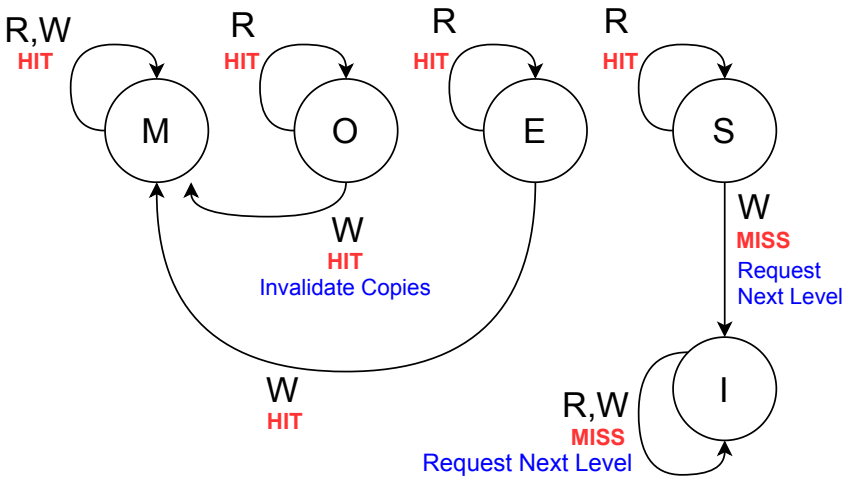


FIGURE 5.14: MOESI coherence protocol when caches process a request

If the request is a write operation, and the line is present in the cache with the *Modified*, *Owned*, or *Exclusive* state, then the response to this request is a hit and the line changes to a *Modified* state. If the line has the *Owned* state, the coherence protocol generates a signal to invalidate any copies of this line with the *Shared* state at any other node. A miss exists if the line is *Shared* or *Invalid*.

If the L1 and L2 caches cannot fulfill the request, the request arrives at the shared bus, and the *snooping process* will check all nodes to see if they have the requested line. This search is limited to the L2 cache level as the uncore implements an inclusive cache policy [71]. Similar to the cache request process, the node that holds the line in a *Modified* or *Owned* state will fulfill the request, update the state of the line, and send messages to other nodes to update the state of the line. The responding node updates the line to the *Owned* state if the request is a read operation. If the request is a write operation, the responding node changes the line to an *Invalid* state and sends a broadcast message to invalidate any copies of the line in other nodes. The message to the requester node has the line in a *Shared* state for a reading request or a *Modified* state for a writing request. Figure 5.15 shows the state machine of this process.

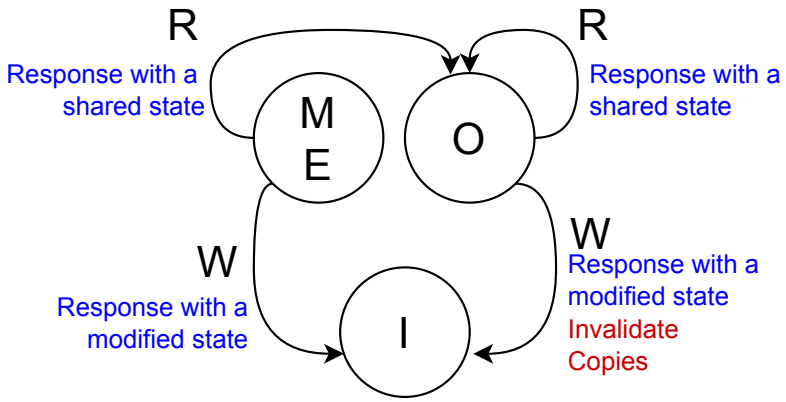


FIGURE 5.15: MOESI coherence protocol in the snooping process

This *snooping process* also manages the broadcast messages to invalidate line copies. The snooper will generate the broadcast messages to invalidate all copies of this line in any node. If a status changes in the L1 cache, an updating message is passed to the L2 cache to maintain the implemented inclusive cache policy. Both L1 and L2 caches process broadcasting messages from snooping. In this way, the process described in section 5.1.1.1 also checks if

a broadcasting message comes from the *snooping process*. Figure 5.16 shows the interface communication of the snooper in all nodes.

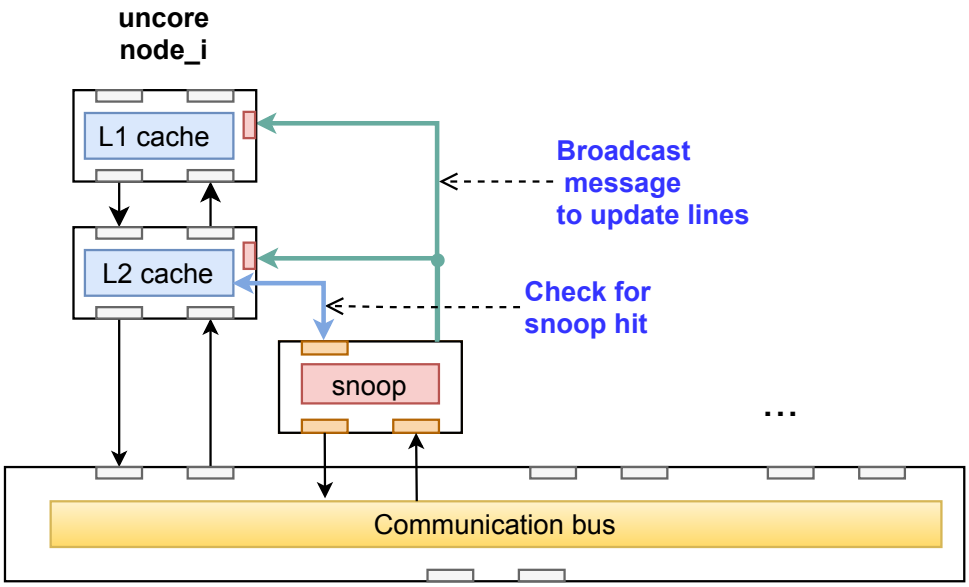


FIGURE 5.16: Communication between snooping process and caches

Appendix A.2 shows the pseudocode of kernels in each uncore process explained in this section.

5.1.4 Computational Cost Analysis for the Target Architecture simulation

To evaluate the performance of the proposed simulator, obtaining an accurate performance calculation would require numerous highly complex parameters. Instead, we analyze the computational complexity of the sequential simulation model under a simplified and hypothetical scenario with multiple memory events and assume the worst-case scenario where the requested data is in the main memory. The sequence of actions for a memory request event from *trace 0* under these conditions is illustrated in Figure 5.17.

The *core model 0* requests (marker ①) the L1 cache, which searches for a hit but misses and passes the request to the L2 cache (marker ②). The L2 cache searches for a hit but also misses, sending the request to the *bus arbiter* (marker ③), which transfers it to the snooper (marker ④). The *snooping process* checks the L2 caches in all n-nodes and finds a miss, sending the request to the *L3 access queue manager* (marker ⑤) and then to the L3 cache (marker

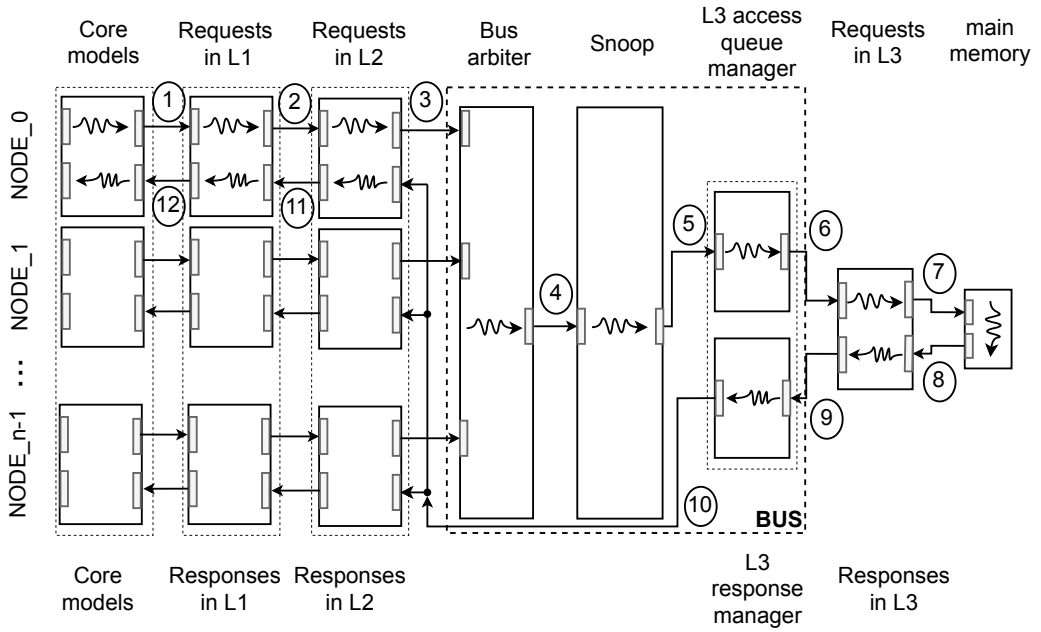


FIGURE 5.17: Processing flow for a requested data

⑥), which also misses. The request arrives at the main memory (marker ⑦), where the simulator finds the data and updates all caches. The response arrives at the L3 cache (marker ⑧), then at the *L3 response manager* (marker ⑨), the L2 cache in *node 0* (marker ⑩), the L1 cache (marker ⑪), and finally, at the corresponding core model (marker ⑫).

The flow in Figure 5.17 is explained in detail by Algorithm 5.1, with lines 2 to 21 outlining the process. The algorithm highlights the frequency of each step for each memory request. Also, it displays the global access rate (calculated as the number of accesses divided by the total number of events simulated), which is dependent on the workload. The computational cost of each process is assumed to be constant to simplify the analysis. The external loop in line 1 allows repetition of this process for r memory requests originating from one or multiple cores.

algorithm 5.1 Sequential		Frequency	Access rate
1:	**for** i = 1 .. r **do**		
2:	coreModel(i)	$r * (1)$	1
3:	**for** j = 1 .. NwL1 **do**		L1Rt
4:	search hit in setLine[i,j]	$r * (NwL1)$	
5:	**for** j = 1 .. NwL2 **do**		L2Rt
6:	search hit in setLine[i,j]	$r * (NwL2)$	
7:	busArbiter()	$r * (1)$	busRt
8:	**for** k = 1 .. n **do**		busRt
9:	**for** j = 1 .. NwL2 **do**		
10:	search hit in setLine[k,j]	$r * (n * NwL2)$	
11:	L3AccessQueue()	$r * (1)$	L3Rt
12:	**for** j = 1 .. NwL3 **do**		L3Rt
13:	search hit in setLine[j]	$r * (NwL3)$	
14:	mainMemory()	$r * (1)$	memRt
15:	**for** j = 1 .. NwL3 **do**		memRt
16:	search place in setLine[j]	$r * (NwL3)$	
17:	L3ResponseManager()	$r * (1)$	L3Rt
18:	**for** j = 1 .. NwL2 **do**		busRt
19:	search place in setLine[i,j]	$r * (NwL2)$	
20:	**for** j = 1 .. NwL1 **do**		L2Rt
21:	search place in setLine[i,j]	$r * (NwL1)$	
22:	coreModel(i)	$r * (1)$	L1Rt

Lines 3 and 4 represent the L1 cache searching process, where a hit is searched for in the set lines one by one (as outlined in Section 5.1.1.1 and Figure 5.8). *NwL1* is the number of searches performed. The L2 cache search process, depicted in lines 5 and 6, is similar to the L1 process, but the number of lines to search is different, with *NwL2* representing the associativity at this level. The *snooping process* in lines 8 to 10 checks the L2 caches in all *n* nodes and runs the process explained in lines 5 and 6 for each cache, with the frequency being the product of the L2 associativity and the number of nodes (*n* * *NwL2*). Lines 12 and 13 show the L3 cache search for a hit, similar to the one for L1 and L2 caches. The modeled main memory has a constant access latency and does not implement any data structure. The response process searches for

a suitable way to place the response data (as outlined in Section 5.1.1.2 and Figure 5.10). Lines 15 and 16 show the loop for this search in the L3 cache, and lines 18 to 21 show the placement process for the L2 and L1 caches. In our hypothetical case of only memory requests, this process is repeated for each node with a memory request event, as shown in the external loop in line 1.

The computational complexity of processing the sequential simulation of these *r* requests is calculated by summing the product of each frequency and its access rate (*L1Rt, L2Rt, busRt, L3Rt, or memRt*), multiplied by its individual computational cost. We will assume that all processes have the exact individual computational cost (*Ks*) to simplify the analysis. Equation 5.1 shows the computational complexity of the simulation.

$$\begin{aligned} C_s = Ks * r * (&1 + NwL1 * L1Rt + NwL2 * L2Rt + 1 * busRt + \\ &n * NwL2 * busRt + 1 * L3Rt + NwL3 * L3Rt + \\ &1 * memRt + NwL3 * memRt + 1 * L3Rt + \\ &NwL2 * busRt + NwL1 * L2Rt + 1 * L1Rt) \\ C_s = Ks * r * \{&1 + (L1Rt + L2Rt) * NwL1 + \\ &[L2Rt + (n+1) * busRt] * NwL2 + \\ &(L3Rt + memRt) * NwL3 * + busRt + \\ &2 * L3Rt + memRt + L1Rt\} \end{aligned} \tag{5.1}$$

In the scenario where all requested data are in the main memory and all access rates equal 1 (Such as a cold cache state) (See Figure 5.17), Equation 5.1 simplifies to Equation 5.2.

$$C_s = Ks * r * [2 * NwL1 + (2+n) * NwL2 + 2 * NwL3 * +6] \tag{5.2}$$

The cost of processing *r* memory requests depends on the associativities of L1, L2, and L3 caches (*NwL1, NwL2, NwL3*), the number of nodes (*n*), and the number of requests (*r*). The cost of this sequential model does not affect whether the requests come from one or multiple cores. However, in the following section, we assume that multiple cores making these r requests simultaneously will improve the parallel simulator's performance.

5.2 Parallelizing the Uncore Simulation

Algorithm 5.1 reveals the inherent parallelism level of the simulation of a memory request event in terms of data and task parallelism across the different levels of the memory hierarchy.

5.2.1 Data Parallelism

Each *for-loop* in lines 2 to 21 in the algorithm represents a process that can be parallelized (such as the hit searching and data placing processes). This inherent data parallelism is dependent on the cache associativity level. Both cache requests and responses and snooping searching processes exhibit this parallelism.

5.2.1.1 Coarse-Grained Data Parallelism: Nodes

When all cores in Figure 5.17 simultaneously request memory, the parallel simulation of the uncore eliminates the external loop in line 1 of Algorithm 5.1, thus reducing the sequential simulation's time cost. This way, the parallel simulation can process all requests simultaneously, just like the target architecture.

5.2.2 Task Parallelism

5.2.2.1 Pipelined Structural Levels in Target Architecture

In Figure 5.17, the cores, L1 cache, L2 cache, communication bus, L3 cache, and main memory have defined functions and data. The target architecture divides the uncore task, enabling pipeline parallelism. Additionally, the cache memory process is split into two tasks (kernels) for handling requests from the higher uncore level and processing responses from the lower uncore level, simplifying cache processing. The communication bus splits its functionality into sub-tasks: *busArbiter*, *snooping process*, *L3AccessQueue*, and *L3ResponseManager*, as described in Section 5.1.2.

5.2.3 Kernel Parallelism Dimensionality (Number of Threads)

From task parallelism, we implement multiple kernels to execute the functionality of each sub-process that results from the task splitting. The level of

TABLE 5.1: Uncore kernel list, dimensionality, and parameters

Uncore kernels	Dimensionality	Parameters
L1request	n-nodes * L1 associativity	- Up and down ports - L1 cache data structure
L2request	n-nodes * L2 associativity	- Up and down ports - L2 cache data structure
L3request	1-nodes * L3 associativity	- Up and down ports - L3 cache data structure
L1response	n-nodes * L1 associativity	- Up and down ports - L1 cache data structure
L2response	n-nodes * L2 associativity	- Up and down ports - L2 cache data structure
L3response	1-nodes * L3 associativity	- Up and down ports - L3 cache data structure
busArbiter	n-nodes	- Up and down ports - Register for data in shared resources
snooping	n-nodes * L2 associativity	- Up and down ports - L2 cache data structure
L3AccessQueue	1	- Up and down ports - L3 access queue
L3ResponseManager	1	- Up and down ports

parallelism of each kernel depends on the data parallelism exhibited by each algorithm. Table 5.1 summarizes the list of uncore kernels, their dimensionality (number of threads), and parameters.

L1request kernel can simultaneously process requests to each L1 cache for all nodes. This kernel has as parameters the L1 cache data structure array for all nodes (See Figure 5.5) and the up and down ports to communicate with their respective cores and L2 caches. Kernel's dimensionality corresponds to the number of nodes multiplied by the L1 associativity. Additionally, it can simultaneously, for each request, check in every way of the set, searching for a hit. Figure 5.18 depicts this dimensionality.

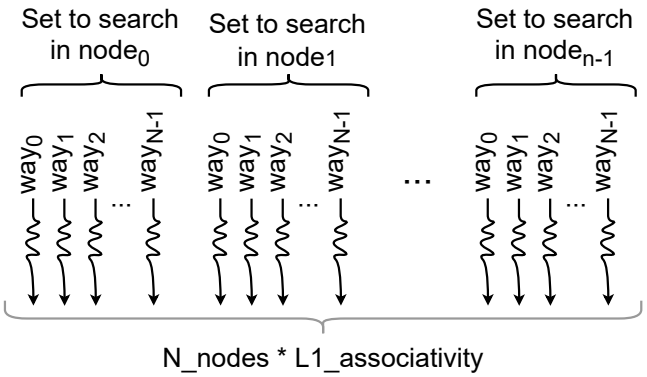


FIGURE 5.18: Kernel parallelism in L1 cache request

The L2 and L3 caches have similar dimensions based on their specific associativity. The *bus arbiter* kernel has a dimension of n, and it concurrently checks requests in all nodes and chooses one randomly. Figure 5.19 displays the dimensions of the entire computer architecture structure. Besides the uncore processes listed in table 5.1, the dimensionality of core models for *memory request* event type, as described in Section 4.4.2, is also shown.

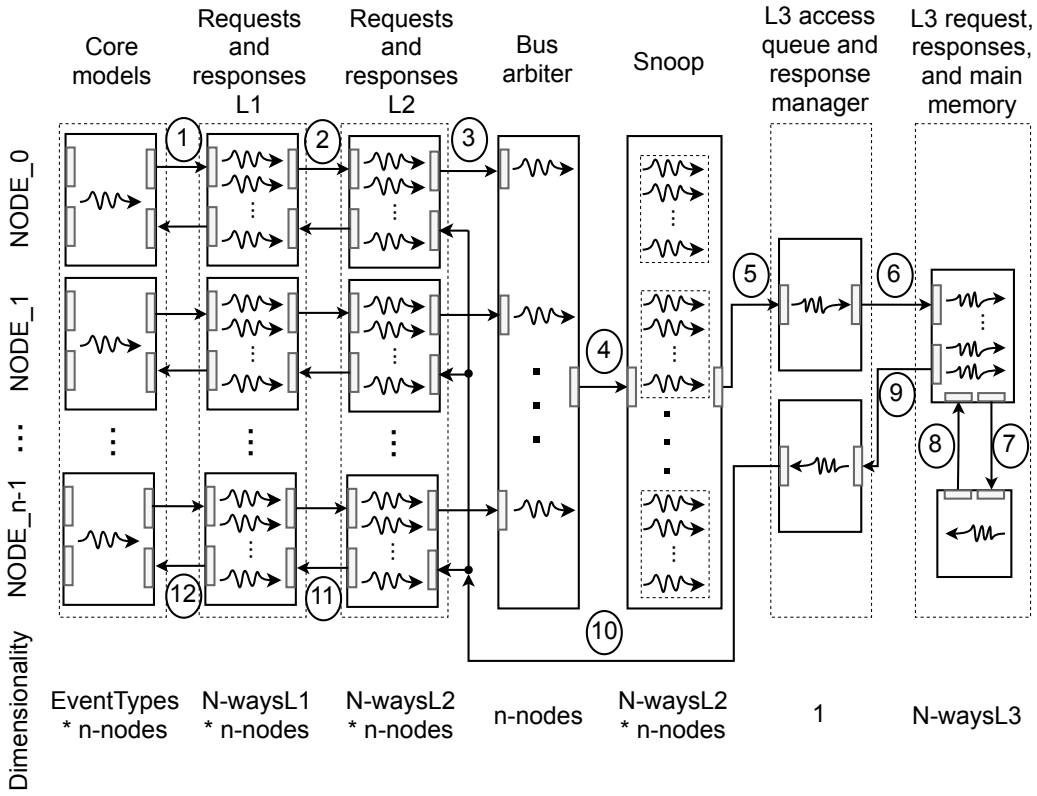


FIGURE 5.19: Parallelism grade in the target computer architecture

Core model kernels simultaneously process the top event in each trace. The best performance in simulation is when all traces simultaneously have memory request events. If the memory request kernel identifies an event of this type, it sends the request to the L1 up-port. Meanwhile, the uncore kernels simulate the operations of the uncore.

Similarly, we programmed kernels to handle responses in each cache level, just like we did to process cache requests. We run the replacement policy outlined in Figure 5.10 with the same degree of parallelism as hit searching. Appendix A.2 explains the algorithms for managing cache requests and responses. Meanwhile, the checking process for n nodes in the bus arbiter is

inherently parallel. Each thread checks for requests in a node, and the bus randomly chooses an available request.

The snoop hit search operation in the *snooping process* can be parallelized like the L2 cache search hit process. This parallelization is because the snooping operation must simultaneously access all nodes and lines in a cache set, as shown in Section 5.1.3 (Figure 5.14). On the other hand, the *L3 access queue* and the *L3 response manager* processes are single-threaded. The algorithms of these processes are presented in Appendix A.3.

Finally, all kernels interact through channels presented and described in Sections 3.4 and 5.1.1.

5.2.4 Computational Cost Analysis for the Parallelized Architecture Simulation

The simulation process calls the kernels that Algorithm 5.2 shows. The parallel simulator's process frequencies do not change with the number of nodes n and cache associativities. However, shared resource frequency still varies based on the number of simultaneous requests. The acceleration is best when all n cores have memory requests simultaneously (the number of requests r should match the number of cores).

CoreModel is a kernel with n threads (the kernel dimensionality is signaled with the triple angular brackets, $<<< n >>>$), each concurrently evaluating a core. Line 2 searches for hits in all L1 caches for all nodes, and each thread concurrently checks if a line is a hit. Line 3 uses the same process as Line 2 but for L2 caches. The *bus Arbiter* randomly selects a request from the L2 cache level (Line 5). Lines 6 to 11 process requests sequentially. The snooper kernel in Line 6 has concurrency similar to Line 3. The *L3AccessQueue* kernel is a single-threaded process that controls access to L3 through a single port. Line 8 searches for a hit in the L3 cache with a parallelism level equal to its associativity (*NwL3*). The *L3 response manager* is a single-threaded process that reads the single output port in the L3 cache and writes the response to the appropriate node. Placement searching in the L2 kernel has the exact dimensions as Line 2. Line 13 follows a similar process. Finally, Line 14 processes responses from the uncore.

algorithm 5.2 Parallel simulation		Frequency	Access rate
1:	coreModel $<<< n >>>$	1	1
2:	searchHitL1 $<<< n * NwL1 >>>$	1	L1Rt
3:	searchHitL2 $<<< n * NwL2 >>>$	1	L2Rt
4:	**for** i = 1 .. r **do**		
5:	busArbiter $<<< n >>>$	r	busRt
6:	snooper $<<< n * NwL2 >>>$	r	busRt
7:	L3AccessQueue $<<< 1 >>>$	r	L3Rt
8:	searchHitL3 $<<< n * NwL3 >>>$	r	L3Rt
9:	mainMemory $<<< 1 >>>$	r	memRt
10:	searchPlaceL3 $<<< 1 * NwL3 >>>$	r	memRt
11:	L3ResponseManager $<<< 1 >>>$	r	L3Rt
12:	searchPlaceL2 $<<< n * NwL2 >>>$	1	busRt
13:	searchPlaceL1 $<<< n * NwL1 >>>$	1	L2Rt
14:	coreModel $<<< n >>>$	1	L1Rt

Equation 5.3 depicts the parallel simulator model's computational complexity (C_{pnp}) without a pipeline in the shared resources. Unlike the computational cost of the sequential process, this equation does not depend on the number of nodes or cache associativities. Just as we assumed a homogeneous computational cost value of *Ks* in the sequential model, we will also assume a homogeneous computational cost value of *Kp* in this model.

$$\begin{aligned} C_{pnp} = Kp * [1 + 1 * L1Rt + 1 * L2Rt + \\ (1 * busRt + 1 * busRt + 1 * L3Rt + 1 * L3Rt + \\ 1 * memRt + 1 * memRt + 1 * L3Rt) * r + \\ 1 * busRt + 1 * L2Rt + 1 * L1Rt] \\ = Kp * [1 + 2 * L1Rt + 2 * L2Rt + busRt \\ (2 * busRt + 3 * L3Rt + 2 * memRt) * r] \end{aligned} \tag{5.3}$$

In our comparative case described in Figure 5.19, all requested data are in the main memory, all access rates are 1, and Equation 5.3 is represented by Equation 5.4.

$$C_{pnp} = Kp * (6 + 7 * r) \tag{5.4}$$

To enhance the performance, the model developed implements a pipeline arrangement in the shared resources. The computational complexity (C_{pp}) with this arrangement is represented by Equation 5.5. Equation 5.6 shows when all access rates are equal to 1 (All data are in the main memory).

$$\begin{aligned} C_{pp} = Kp * [1 + L1Rt + L2Rt + busRt + busRt + L3Rt + L3Rt + \\ memRt + memRt + L3Rt + (r-1) * busRt + \\ busRt + L2Rt + L1Rt] \\ = Kp * [1 + 2 * L1Rt + 2 * L2Rt + 2 * busRt + \\ 3 * L3Rt + 2 * memRt + r * busRt] \end{aligned} \tag{5.5}$$

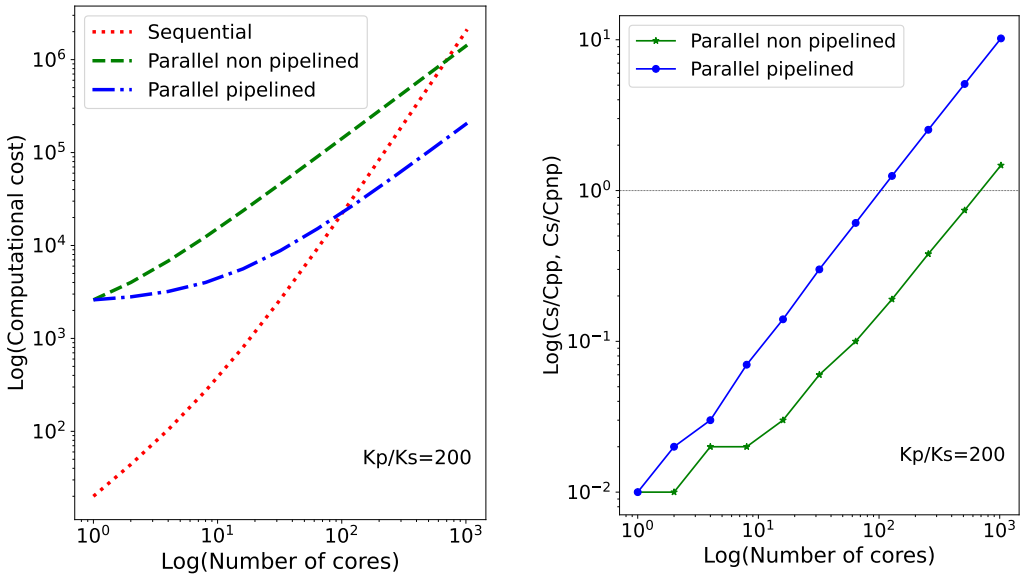
$$C_{pp} = Kp * (12 + r) \tag{5.6}$$

We estimate the performance of sequential and parallel models using Equations 5.2, 5.4, and 5.6, with the number of cores varying from 1 to 1024. The scenario assumes all cores are making memory requests simultaneously, all events are memory requests, and the requested data is in the main memory. The model uses an associative value of 2 (*NwL1=NwL2=NwL3=2*) with *Ks* set to 1. The cost of processes in the parallel model is higher than in the sequential model. The value of *Kp* depends on multiple factors, and to reflect real-world situations, this model must incorporate them:

- Lower frequencies in GPU vs. CPU (2× to 3×).
- Lower performance to process control flow.
- Kernel launch overhead.
- Unbalanced pipeline.
- The assumption that all cores process memory requests simultaneously at all times is highly unrealistic and must consider workload characteristics and memory request rates.

Determine a value for *Kp* is complex indeed to these dependencies. To calculate the behavior of this simplified model to estimate the behavior of our parallel simulator, we set *Kp* to 200. This model will be compared with results in Chapter 6.

Figure 5.20a displays the computational complexity of the sequential model (*Cs*), the parallelized without pipeline (*Cpnp*), and the parallel with the pipeline (*Cpp*). This figure demonstrates that the sequential model (*Ts*) has a lower complexity by two orders of magnitude for a low number of cores (*Cpnp* and *Cpp*). Nevertheless, globally, as the number of cores grows by two orders of magnitude, *Ts* increases by three orders of magnitude, *Cpnp* increases by two orders of magnitude, and *Cpp* increases by only one. This fact highlights the advantage of the parallel versions for a more significant number of cores. Figure 5.20b shows the ratios *Cs*/*Cpp* and *Cs*/*Cpnp*. This figure indicates that under this simplified scenario, the pipelined version surpasses the sequential model near 100 cores, while the non-pipelined version surpasses it above 700 cores.



(A) Computational cost comparison

(B) Speedup comparison

FIGURE 5.20: Comparison between sequential and parallel model

5.3 Summary

This chapter presented the parallelization of the uncore simulator to implement it on a GPU and how we model an uncore in a parallel fashion. We establish the uncore architecture and each of its components. We established the data and task parallelism in uncore and set the parallelism dimensionality. Then, we implement the functionality of the modeled uncore simulator in

the GPU. We analyzed the speedup of the parallel simulator compared with a sequential version.

Chapter 6

Experimental Results

In this chapter, the proposed GPU-based simulator undergoes evaluation. The evaluation aims to estimate the speedup and accuracy of the GPU-based simulator compared to a sequential reference simulator based on GEM5 running in the CPU. Accuracy metrics include simulation cycles and hits/misses for each cache on the target architecture. The chapter details the experimental setup, the reference simulator used for comparison, and the evaluation metrics.

6.1 Experimental Setup

Our assessment of the parallel simulation strategy involves simulating multiple architectures with various workloads and comparing the performance and simulation outcomes to those of similar simulations in a reference simulator. We use the following metrics for evaluation:

- Speedup: We compare the simulation times of both simulators and calculate the ratio between the reference time and our simulator time, such as Equation 6.1 shows.

$$Speedup = \frac{T_{REF}}{T_{GPU}} \tag{6.1}$$

- Accuracy: We measure simulation results in terms of simulation cycles and misses at each cache level. To conduct the evaluation, we calculated the relative error by dividing the absolute error by the mean of the two values being compared [101], as demonstrated in Equation 6.2.

$$Relative\ error = \frac{\mid Value_{REF} - Value_{GPU} \mid}{\frac{1}{2}(Value_{REF} + Value_{GPU})} \times 100 \tag{6.2}$$

Our experiment simulates the architecture of Figure 5.1 for a number of processors from 1 to 128. We configure the core model's average number of cycles per integer and floating operations. Besides, for the uncore, we configure latencies for each cache level and the snooping process, cache sizes, associativity for each cache level, and the size line. Table 6.1 shows the configuration settings for these simulations.

TABLE 6.1: Configuration values

	LATENCIES (Cycles)	SIZES (Bytes)	ASSOC.	SIZE LINE	INTEGER INSTR. (Cycles)	FLOATING INSTR. (Cycles)
L1	4	64K	2	64	-	-
L2	12	2M	2	64	-	-
L3	32	8M	2	64	-	-
Snoop	12	-	-	-	-	-
Core model	-	-	-	-	1	5

The sequential simulator used as a reference for comparative purposes includes the original Synchrotrace implementation connected to a GEM5-based uncore. We configure this uncore model to be as similar as possible to our GPU-based uncore, but some differences remain. Our simulator lacks write-back buffers, while GEM5 allows a minimum of one write-back buffer but does not allow for their elimination. Our GPU-based uncore uses NRU replacement policy, whereas the GEM5-based uncore provides LRU and random policies. For consistency, we choose the LRU policy. There is another difference in the event-driven scheduler model. Our scheduler works with the core model trace events. The reference simulator's scheduler also includes uncore events. All these differences are potential error sources, but this is the closest possible model we can have to ours.

To evaluate the simulator's performance on architectures with different numbers of cores, it is necessary to obtain traces of test programs that run with the same number of threads as the number of cores. Each core in the simulated architecture will process one trace. The workload used in the evaluation is the Splash2 benchmark suite [109] [87]. Benchmarks are simulated

TABLE 6.2: Splash2 benchmarks description [109] [87]

Name	Description
barnes	-N-body interaction problem using the Barnes-Hut method. -Implications of hierarchical N-Body methods for multiprocessor architectures
fmm	-N-body interaction problem using the adaptive Fast Multiple Method. -The communication patterns lack structure. There is no effort to distribute particle data in the main memory intelligently.
ocean-cp (contiguous partitions)	-Large-scale ocean movements based on eddy and boundary currents. -Subgrids improve the communication computation ratio. -Multi-Level adaptive solutions to boundary-Value problems.
radiosity	-Compute the equilibrium distribution of light in a given scene. -The computation's structure and the data access patterns are extremely irregular. Each processor has a separate task queue, and task stealing balances the load.
raytrace	-Renders a three-dimensional scene using ray tracing. -The access patterns to data are extremely irregular.
volrend	-Rendering a three-dimensional volume onto a two-dimensional image plane using an optimized ray casting technique. -Volume Rendering on scalable shared-memory MIMD architectures. -Data accesses are input-dependent and irregular.
water-nsq	-Molecular dynamics N-body problem optimized up to several hundred to a coupe of thousand molecules. -Forces algorithm has a complexity of $O(n^2)$. -Each process operates local data, and updates the shared copy once at the end.
water-spatial	-Molecular dynamics N-body problem optimized for large systems. -Forces algorithm has a complexity of O(n). -The change of cells for a molecule, results in communication.
ocean-ncp (non contiguous partitions)	-Large-scale ocean movements based on eddy and boundary currents. -Multi-Level adaptive solutions to boundary-value problems.
cholesky	-Factorization a sparse matrix into a lower triangular matrix and its transpose. -Large communication computation ratio. -Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization.
fft	-FFT algorithm optimized to minimize interprocessor communication. -To avoid hotspotting data are communicated in a staggered fashion.
lu-cb (contiguous blocks)	-Factoring a dense matrix into the product of a lower triangular and an upper triangular matrix. -Exploit temporal locality and reduce communication. -The performance advantages of integrating block data transfer in cache-coherent multiprocessors.
lu-ncb (non contiguous blocks)	-Factoring a dense matrix into the product of a lower triangular and an upper triangular matrix. -The performance advantages of integrating block data transfer in cache-coherent multiprocessors.
radix	-Sorting Algorithm. -Keys permutation requires all-to-all nodes communication . -A comparison of sorting algorithms for the connection machine.

completely without any fast-forwarding approach. This suite is used by multiple works in the computer architecture evaluation [48] [19] [2] [15] [69] [57]. Table 6.2 depicts a description of these benchmarks.

However, some programs in the suite generated errors during trace acquisition, leading to their exclusion from the experiment. In other cases, trace sizes were excessively large, requiring their removal to conduct experiments and numerous simulations within a reasonable time frame. Table 6.3 lists the excluded programs in the suite and the reasons for their exclusion.

TABLE 6.3: List of excluded benchmarks

BENCHMARK	REASON
radiosity, raytrace	The generated traces for this program were approximately 70GBs, which is 10 times larger than the traces for the other programs.
ocean-cp, ocean-ncp water-spatial, lu-cb	The maximum number of threads for which the traces could be generated and simulated was 64. For 128 threads, the generation process presented several issues.
fmm	Executing the program and generating the traces was challenging, even for a single thread.

We obtained these traces by running the benchmarks on an Intel Xeon Gold 6152 @2.10GHz system for 1 to 128 threads. Synchrotrace's trace generation stage generates one trace per thread during the multithreaded execution of these benchmarks. The performance advantage of the GPU simulation model over a sequential simulation model is analyzed. The results were used as a reference to evaluate our proposed GPU simulation model. We simulated traces generated in the previous step using the reference simulator on a system with an Intel Xeon Gold 6152 @2.10GHz. We performed the parallel simulator tests on an NVidia QUADRO P5000 GPU with 20 Streaming Multiprocessors (SM), 16 GB of RAM, 128 CUDA cores per SM, and 64K 32-bit registers per SM.

We used the following software versions to generate the traces and run the simulations: Ubuntu 16.04.7, g++ 5.4, python 2.7, NVIDIA 418.56, and CUDA 10.1.

6.2 Evaluation of GPU-based Simulator

We assess the simulation times and accuracy of sequential, and GPU parallelized simulations and analyze the impact of the number of nodes on their performance in the multi-core architectures under study.

6.2.1 Scaling the GPU-based Simulation Speedup

Figure 6.1 compares the GPU-based and sequential simulators' scaling performance.

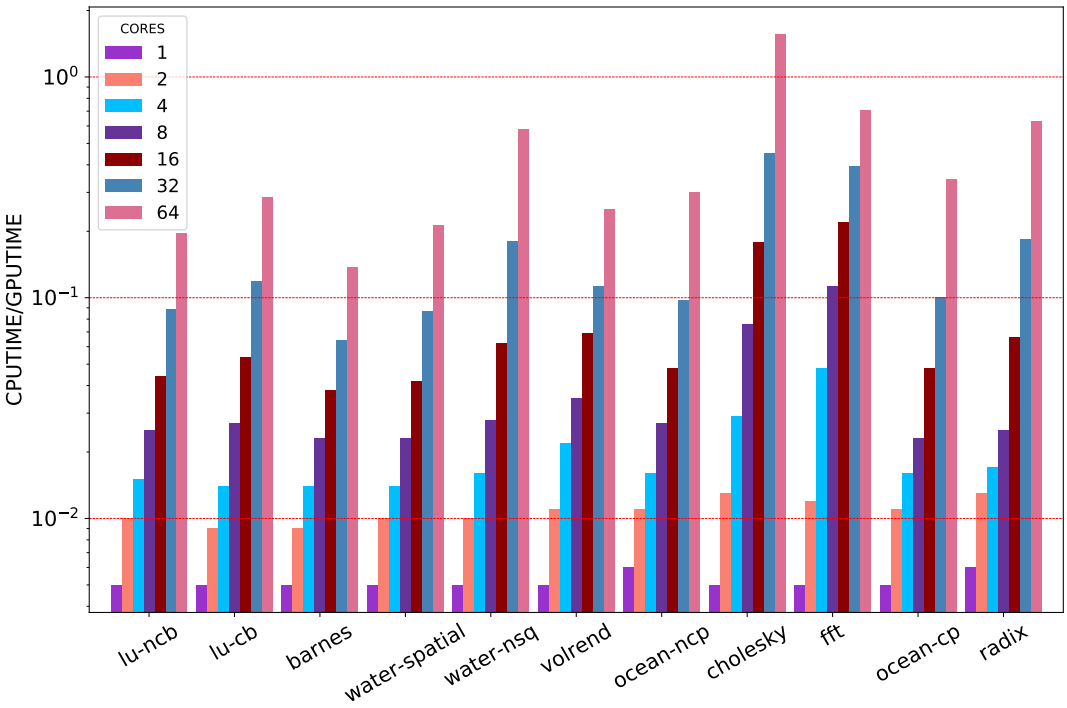


FIGURE 6.1: GPU simulation scaling

The GPU-parallelized simulator is significantly slower than the sequential simulator, which is two orders of magnitude slower for single-core architecture simulation. However, this slowdown decreases as the number of nodes in simulated multi-core architecture increases. This slowdown reduction is consistent with each increase of nodes in the architecture. The trend shows that the performance gap in our GPU simulation increases concerning the CPU version when we increase the number of benchmark threads to simulate. Our simulation exhibits a better performance in benchmarks with a high communication-computation ratio and low synchronization loads, such as cholesky, fft, water-nsq, or radix. Specifically, for 64 threads, the cholesky benchmark in the GPU-based simulation demonstrates a 1.4× speedup over the sequential simulator.

The original Synchrotrace report [89] shows that this model scales to 64 threads. We analyzed the scaling with 128 threads in our GPU model, but some benchmarks started failing from the initial stage of trace generation. The Synchrotrace model and tools become more brittle/unstable as threads increase.

Figure 6.2 shows the simulation results of a subset of benchmarks using 128 threads, indicating the trend in scaling for speedup. The water-nsquared benchmark achieved a speedup of 1.7×, cholesky 4.3×, fft 1.2×, and radix

2.1×. This trend in speedup is consistent with the results presented in Figure 6.1.

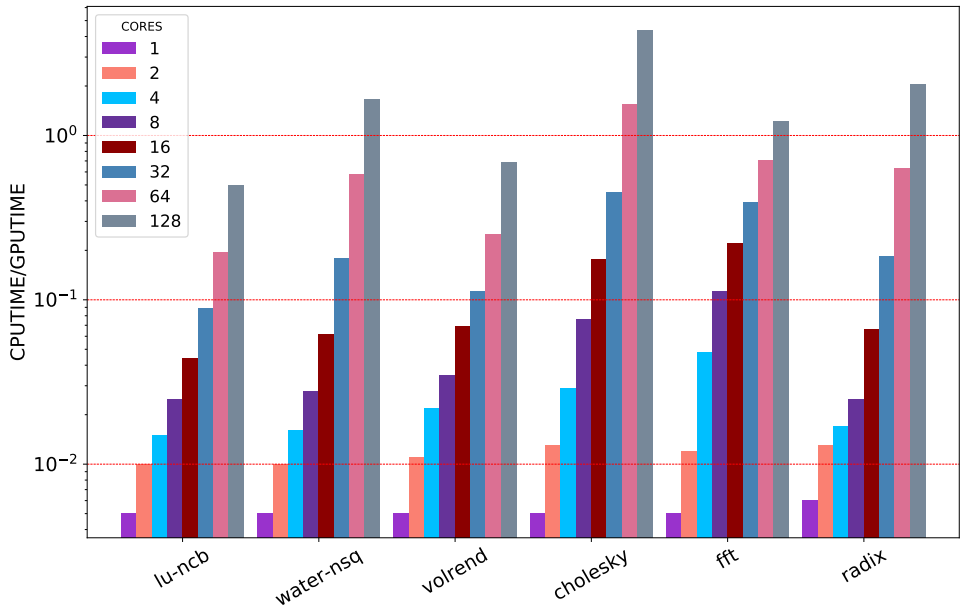


FIGURE 6.2: GPU simulation scaling until 128 threads

The graph in Figure 6.3 illustrates the theoretical acceleration (dashed line) based on the model described in Section 4.4.2 compared with the actual acceleration (continuous lines). This model shows a speedup from 100 cores. Although we simulated until 128 cores, the trend to growth is clear to a greater number of cores. However, a factor that would saturate the scaling is the capacity of GPU resources.

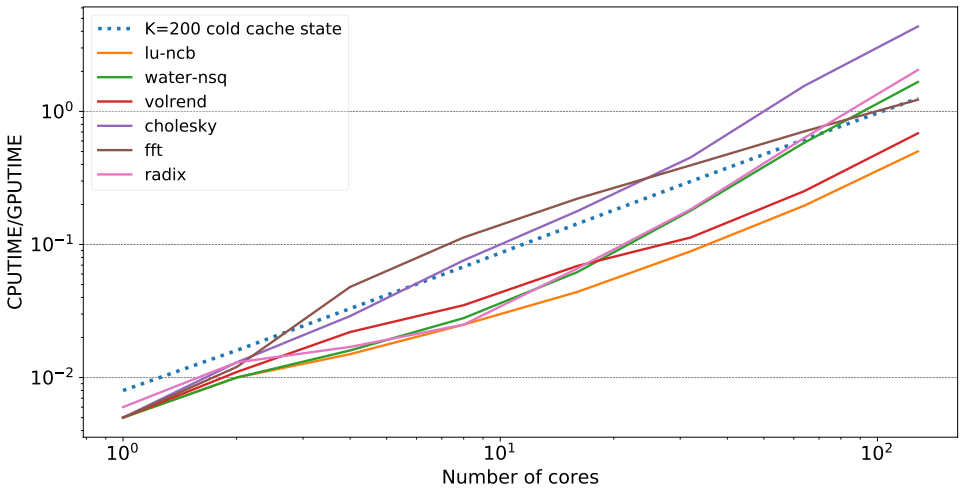


FIGURE 6.3: Theoretical and experimental speedup comparison

6.2.2 Analysis of Speedup Results

The type of workload evaluated has a significant impact on the resulting performance value. A report analysis in [109], summarized in Table 6.4, reveals that workloads such as cholesky, fft, and radix have a higher uncore traffic (Bytes per Instructions) than barnes, lu, water-spatial, and volrend, which have a lower traffic count. We found a strong correlation between workload traffic and simulation performance (6.1). However, there are exceptions, such as water-nsq and ocean. Water-nsq has an unusually high number of write-backs that could reduce the performance of the sequential simulator. In contrast, Ocean has traffic levels similar to fft but an exceedingly high number of synchronization barriers, which could degrade parallel simulator performance.

TABLE 6.4: Traffic and Synchonization metrics in Splash2 benchmarks

	fft	ocean	cholesky	radix	lu cb,ncb	barnes	water nsq	water spatial	volrend
Traffic	0.9	0.8	0.65	0.76	0.2	0.17	0.2	0.09	0.08
Barriers	6	364	3	10	66	8	10	10	15

Further analysis can be performed by comparing the Millions of Instructions Per Second (MIPS) between sequential and parallel simulators. As depicted in Figure 6.4, the average MIPS is shown for each number of simulated cores. The graph clearly illustrates that the sequential simulator experiences rapid performance degradation as the number of cores increases. In contrast, the parallel simulator's performance doesn't decline with increasing cores. Moreover, the parallel simulator outperforms the sequential one by around 100 cores, and the trend suggests that the speedup can become more prominent with an increasing number of cores.

Akram *et al.* [5] have reported the average simulation time for some state-of-the-art simulators, all sequential, when simulating 500 million instructions in a single core and an uncore with L1, L2, and L3. We calculated the MIPS for each simulator using these results, summarized in Table 6.5. Although they don't simulate systems identical to our experimental setup, they give us an idea of the range of MIPS obtained in these simulators. It's worth noting that the MIPS achieved by these simulators falls within the range obtained in both the sequential and parallel Synchrotrace simulator. Despite having low MIPS for only one core, our parallel simulator would still represent a

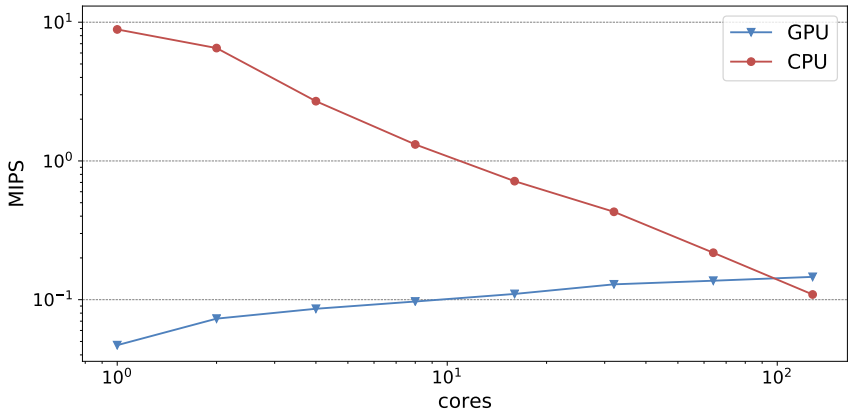


FIGURE 6.4: MIPS comparison

speedup of 2.35× compared to the GEM5 simulator. We should highlight that these are sequential simulators that, similar to the sequential Synchrotrace, can be expected to experience rapid degradation in MIPS performance as the simulated architecture includes more cores, which is not the case with our parallel simulator.

TABLE 6.5: MIPS in state-of-the-art sequential simulators for one-core

	GEM5	MARSSx86	Sniper	Zsim	Sequential Synchrotrace	Parallel simulator
Instructions (10^6)	500	500	500	500	28490	28490
Sim. Time ($10^3 sec$)	24.3	0.79	0.32	0.15	3.2	600
MIPS	**0.02**	**0.63**	**1.57**	**3.36**	**8.9**	**0.047**

Concerning enhancing the performance of the parallel simulator, a key to maximizing computational efficiency on the GPU in the simulation model is to have kernels that utilize multiple warps and exploit the GPU's execution model. However, when simulating architectures with a limited number of cores, the GPU simulation becomes less effective because the dimensions of several kernels of the simulation model are not even enough to fill the warps—increasing parallelism in the application results in better GPU efficiency than the CPU, allowing the kernels to fill more warps. While the simulation strategy only slightly outperforms CPU performance for 128 threads, it has the potential to significantly accelerate the simulation of architectures with hundreds of cores, meeting the simulation requirements for server processors that could contain 512 cores and more.

6.2.3 Accuracy in the GPU-Based Simulator

The obtained results showcase the potential of the proposed simulation model to accelerate the detailed simulation of uncore in architectures with hundreds of cores. An additional challenge faced in this work was establishing the reference model to assess the accuracy and performance of the developed model. While the models are not precisely the same, and some variation in simulation results is expected, the primary objective of this comparison is to illustrate improved performance with comparable functional outcomes.

We assess the accuracy of the simulation clock cycles metric in the GPU-parallelized simulator by comparing it with the reference simulator. Figure 6.5 demonstrates the relative errors for each benchmark in each architecture. The simulator's relative error is below 20%, except for the lu-cb, lu-ncb, and barnes benchmarks in 2-core architectures. These benchmarks for two threads have unusually high replacement and writeback rates (see [109]), which may explain these peaks in the figure. The difference in implementation of the event scheduler, replacement policies (NRU vs. LRU), and write-back buffers (victim buffer) impacts simulation cycle metrics in these benchmarks. However, simulations for other benchmarks show a lower than 10% relative error, even less than 5%. In the best case, for the benchmarks water-nsq, ocean-ncp, cholesky, fft, ocean-cp, and radix.

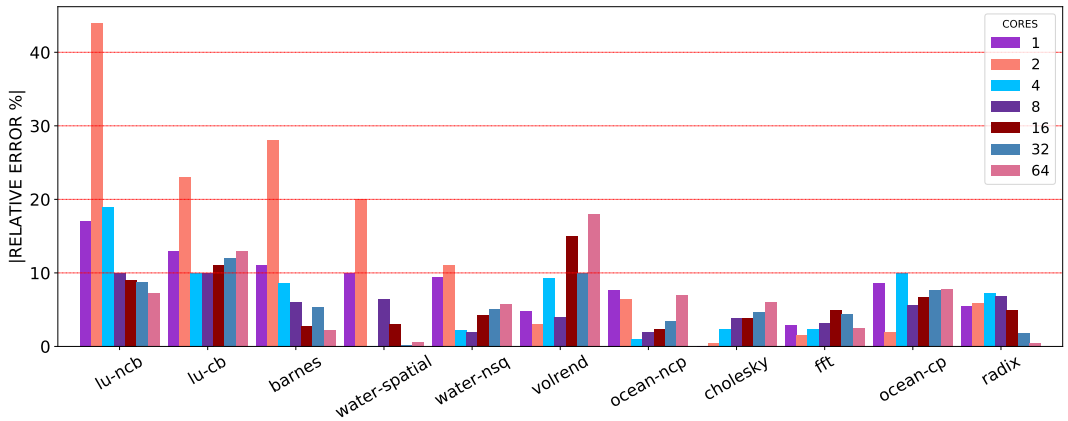


FIGURE 6.5: SimTicks detailed accuracy

Besides simulation cycle accuracy, we analyze the accuracy of each cache level's behavior and compare the number of hits and misses with the reference simulator. While the error in the number of hits in the L1 and L2 cache levels is negligible, in cache L3, this metric is significant, with errors as high as 80%. These errors occur when the number of L3 hits is much smaller

than the total number of accesses at this level, making any difference in L3 hits very noticeable. In contrast, the accuracy of the number of cache misses is more stable across all cache levels. In some cases, we observe an error between 10% and 20%, but most cases have an error lower than 5%, and in several cases, the error is close to zero. Figure 6.6 illustrates the accuracy of the cache behavior in terms of misses for each benchmark and each cache level.

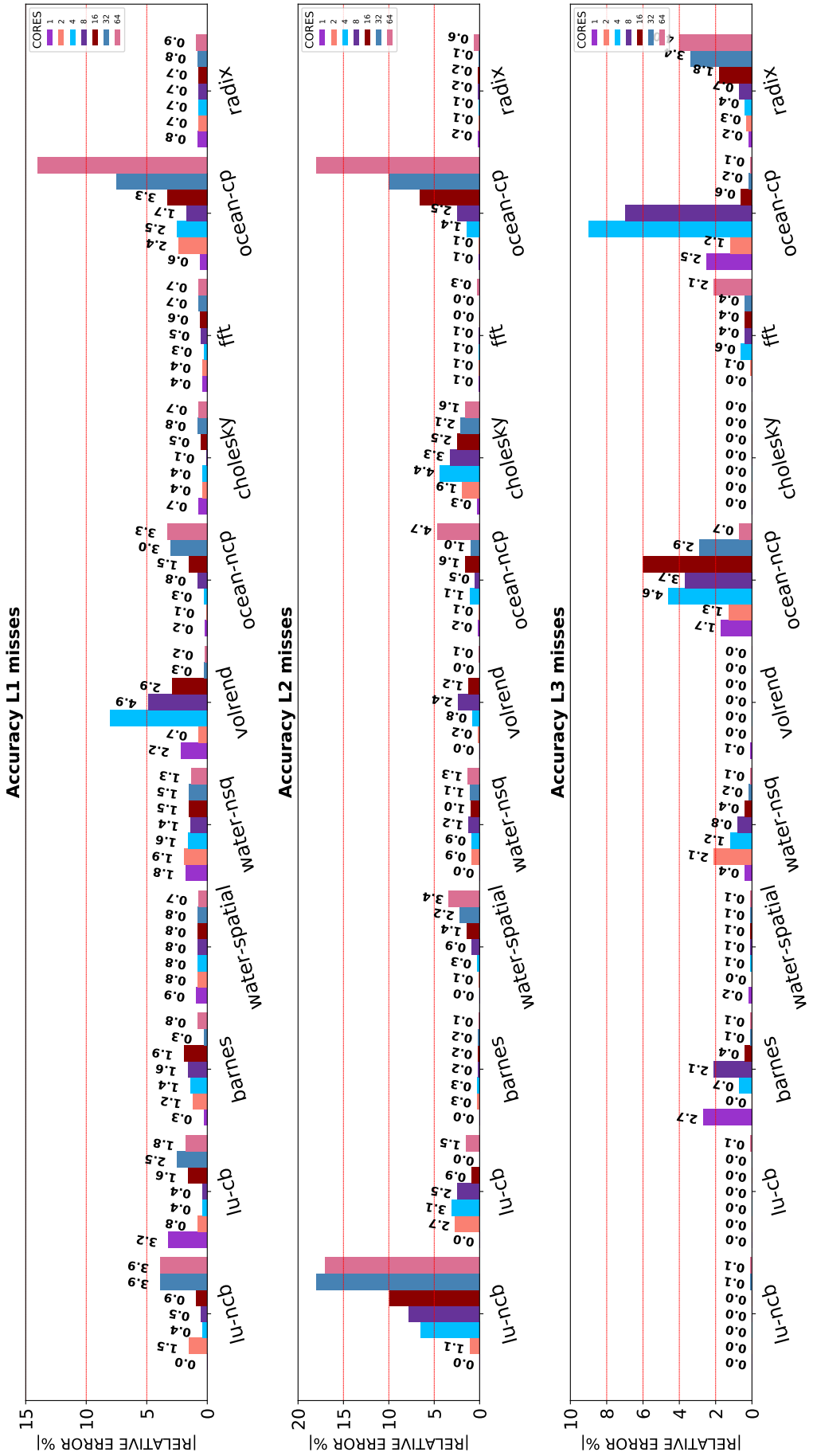


FIGURE 6.6: Detailed misses cache Accuracy

Chapter 7

Conclusions and Future Work

This thesis has presented a new methodology for implementing computer architecture simulators, focusing on the uncore. The methodology combines approximated simulations of the cores using behavioral core models with a detailed model of the uncore. Both models are simulated on a GPU using an event-driven simulation strategy. We have described the strategy for implementing the interaction between the core model and the uncore and the parallelization approaches applied to leverage the GPU's computational power. This simulation methodology is a valuable addition to the toolbox of computer architecture simulation.

During the development of this work, we extensively studied the core models available in the state of the art, allowing us to establish important qualitative selection criteria for developing our methodology. The Synchrotrace model was selected as our reference core model due to its multithread support and open-source availability. We modified the Synchrotrace traces, resulting in a modified parallel Synchrotrace model that processes simpler and more regular events, making it convenient for GPU execution. However, these modifications increased the number of events in the new traces compared to the original ones.

A significant portion of our work was dedicated to developing a detailed parallel model of the uncore, including the most important components in a modern multicore processor memory hierarchy. We applied several parallelization strategies to minimize simulation time. Different combinations of kernel configurations lead to different performance levels. Therefore, there is still room for improvement.

The simulation results showed that achieving speedup with respect to the reference simulator is only possible when simulating architectures with hundreds of cores. Our simulator strategy exhibits a strong workload dependency, showing better speedup in benchmarks with high communication-computation ratios and low synchronization loads.

We developed an analytical performance model for the simulator's execution on the GPU and another model for its execution on the CPU. We used both models to compute the theoretical speedup and identified the limitations of the proposed model. The model aids in forecasting the performance trend of our simulator.

Regarding the accuracy of simulation, most results demonstrate a relative error of less than 10%. However, in a 2-core architecture, up to 47% of the errors were significant, particularly in benchmarks involving high replacement and writeback rates. These errors revealed the model's sensitivity to specific workloads, probably due to disparities between the reference model and our own.

This work has demonstrated that GPUs can be used as host platforms for computer architecture simulation when simulating architectures with hundreds of cores and beyond. Today, we have processors with over a hundred cores in the market, and the number continues to increase with every new generation. Together with the number of cores, the uncore's complexity increases too, and thus the need for better tools for computer architecture simulation. The computational power of GPUs increases with every new generation. This work presented a methodology to harness that computing power to benefit computer architecture simulation.

Overall, our methodology could represent a contribution to the field of computer architecture simulation. However, there is still room for improvement, and future work could focus on improving the simulator's performance by means of more efficient use of the CUDA programming model. The development of a DSL to abstract the complexity of GPU programming is also a need for the broad adoption of GPUs as simulation platforms by computer architect engineers.

7.1 Future work

Possible research directions to enhance the computer architecture simulation field are identified from this work's achievements, results, and limitations.

Concerning enhancing the simulator performance:

1. **Extending the core model:** A core model that supports Out of Order (OoO) architecture can better utilize the simulator resources. An OoO model increases the number of simultaneous requests to uncore, resulting in increased speedup compared to a sequential simulator.

Concerning enhancing the simulator accuracy:

1. **Enhance the detail of the uncore model:** implementations that include write-back buffers and victim caches allow a simulator to be similar to reference simulators and get more accurate results.

Concerning extending the simulation versatility:

1. **Extend uncore model:** Design space exploration can require other replacement policies and other coherence protocols. Furthermore, the modeled uncore can be extended to support directory-based computer architectures and to implement another network topology to replace the shared bus implemented in this work.
2. **Domain-Specific Language (DSL):** The complexity of GPU programming represents a drawback for computer architects to adopt GPUs as simulation platforms. To address this, a DSL close to the computer architecture designer can abstract the GPU programming model to implement the computer simulator. It should be possible to match the proper granularity of simulator constructs with GPU hardware components through the DSL. Simulator data structures should also be well-defined and scheduled to use GPU memory efficiently.

Appendix A

CUDA Kernels Algorithms

A.1 Core model Kernels

The following appendix includes pseudo-codes of the kernels for each event type core model discussed in Section 4.4.2. These kernels require the event traces (TEQ in Figure 3.3) and an array of counters of simulated events (counterEvs) for each trace. Additionally, the kernels take in the global simulation cycles and the launch cycles array (refer to Core Model Timing Controller in Section 4.4.3). For kernels involving mutexes and conditional variables, the mutexes array (mutexArray) and conditional variables matrix (condVbleMatrix) (refer to Figure 4.2) are also required.

Algorithm A.1 CORE CYCLES EVENTS

```
1: function CORE_CYCLES(TEQ, counterEvs, globalSimCycle, launchCycles)
2:     traceIdx = treadIdx.x + blockIdx.x * blockDim.x
3:     thisEventCount = counterEvs[traceIdx]
4:     thisEvent=TEQ[traceIdx][thisEventCount]
5:     if thisEvent.Type = CORE_CYCLES then
6:         event_Cycles = thisEvent.IOPS * CYC_INT_INST +
                   thisEvent.FLOPS*CYC_FLT_INST
                                ▷ CYCLES PER INST are stored in the CONSTANT MEM
7:         launchCycles[traceIdx]=globalSimCycle + event_Cycles
8:         counterEvs[traceIdx]++                                  ▷ commit action
```

Algorithm A.2 MUTEX UNLOCK EVENTS

```
1: function MUTEX_UNLOCK(TEQ, counterEvs, globalSimCycle, launchCycles, mutexArray)
2:     traceIdx = treadIdx.x + blockIdx.x * blockDim.x
3:     thisEventCount = counterEvs[traceIdx]
4:     thisEvent=TEQ[traceIdx][thisEventCount]
5:     if thisEvent.Type = MUTEX_UNLOCK then
6:         mutexArray[thisEvent.MUTEX]=UNLOCKED
7:         event_Cycles = 1
8:         launchCycles[traceIdx]=globalSimCycle + event_Cycles
9:         counterEvs[traceIdx]++                                   ▷ commit action
```

Algorithm A.3 CONDITIONAL VARIABLE BROADCASTING EVENTS

```
1: function CNDVBL_BROADCASTING(TEQ, counterEvs, globalSimCycle, launchCycles, setCndVbleRqstPort)
2:     traceIdx = treadIdx.x + blockIdx.x * blockDim.x
3:     thisEventCount = counterEvs[traceIdx]
4:     thisEvent=TEQ[traceIdx][thisEventCount]
5:     if thisEvent.Type = CNDVBLE_BROADCASTING then
6:         setCndVbleRqstPort=msg(thisEvnt.COND_VBLE, SET_BROADCASTING)
7:         event_Cycles = 1
8:         launchCycles[traceIdx]=globalSimCycle + event_Cycles
9:         counterEvs[traceIdx]++                                   ▷ commit action
```

Algorithm A.4 BROADCASTING IN CONDITIONAL VARIABLE MATRIX

```
1: function BRDCST_CNDVBLMATRIX(condVbleMatrix, setCndVbleRqstPort)
2:     traceIdx = treadIdx.x + blockIdx.x * blockDim.x
3:     if setCndVbleRqstPort.Type = SET_BROADCASTING then
4:         COND_VBLE=setCndVbleRqstPort.COND_VBLE
5:         condVbleMatrix[COND_VBLE][traceIdx]=1
```

Algorithm A.5 MUTEX LOCK EVENTS

```
1: function MUTEX_LOCK(mutex_unlock)TEQ, counterEvs, globalSimCycle, launchCycles, mutexArray
2:     traceIdx = treadIdx.x + blockIdx.x * blockDim.x
3:     thisEventCount = counterEvs[traceIdx]
4:     thisEvent=TEQ[traceIdx][thisEventCount]
5:     if thisEvent.Type = MUTEX_LOCK then
6:         if mutexArray[thisEvent.MUTEX]=UNLOCKED then
7:             mutexArray[thisEvent.MUTEX]=LOCKED
8:             event_Cycles = 1
9:             launchCycles[traceIdx]=globalSimCycle + event_Cycles
10:            counterEvs[traceIdx]++                               ▷ commit action
11:        else
12:            launchCycles[traceIdx] = MAX_UINT64;                 ▷ waiting action
```

Algorithm A.6 DEPENDENCE EVENTS

```
1:  function DEPENDENCE(TEQ, counterEvs, globalSimCycle, launchCycles)
2:      traceIdx = treadIdx.x + blockIdx.x * blockDim.x
3:      thisEventCount = counterEvs[traceIdx]
4:      thisEvent=TEQ[traceIdx][thisEventCount]
5:      if thisEvent.Type = DEPENDENCE then
6:          traceOfDependence = thisEvent.TrcDep
7:          eventOfDependence = thisEvent.EvntDep
8:          if eventOfDependence<counterEvs[traceOfDependence] then
9:              event_Cycles = 1
10:             launchCycles[traceIdx]=globalSimCycle + event_Cycles
11:             counterEvs[traceIdx]++                          ▷ commit action
12:         else
13:             launchCycles[traceIdx]=MAX_UINT64;              ▷ waiting action
```

Algorithm A.7 MEMORY EVENTS

```
1:  function MEM_REQUEST(TEQ, counterEvs, globalSimCycle, launchCycles, uncore-
    Ports)
2:      ndeIdx = treadIdx.x + blockIdx.x * blockDim.x
3:      thisEventCount = counterEvs[traceIdx]
4:      thisEvent=TEQ[traceIdx][thisEventCount]
5:      if thisEvent.Type = (READ or WRITE) then
6:          if thisEvent.State!=WAITING then
7:              uncorePorts[uncorePorts]=msg(thisEvent.Type, thisEvent.Address)
8:              thisEvent.State=WAITING
9:              launchCycles[traceIdx] = MAX_UINT64
                                          ▷ Uncore will update this value
10:         else                                                ▷ waiting state
11:             thisLaunchCycle=launchCycles[traceIdx]
12:             if thisLaunchCycle<=globalSimCycle then
                                          ▷ Uncore resolved the request?
13:                 counterEvs[traceIdx]++                      ▷ commit action
```

A.2 Cache memory Kernels

The arguments passed to the uncore kernels consist of structured data that represent the cache information, as well as a structure that represents the communication ports for each level, as described in Section 5.1.

Algorithm A.8 Requests in cache L1

```
1:  function CACHEL1_REQUEST(cchL1, LowPortsL1, UpPortsL1)
2:      ndeIdx = threadIdx.y
3:      thisRqst=UpPortsL1[ndeIdx]
4:      if thisRqst.RqstType=READ then
5:          lnIdx=threadIdx.x
6:          numSet=getNumSet(thisRqst.Address)
7:          thisTag=getTag(thisRqst.Address)
8:          thisLine=cchL1[ndeIdx][numSet][lnIdx]
9:          if thisLine.TAG=thisTag && thisLine.State!=INVALID then
10:             hitProcessing()                          ▷ REPLACEMENT POLICY update
11:         else
12:             LowPortsL1[ndeIdx]=thisRqst ▷ ByPass request to nextLevel, use LowPortsL1
13:     else if thisRqst.RqstType=WRITE then
14:         lnIdx=threadIdx.x
15:         numSet=getNumSet(thisRqst.Address)
16:         thisTag=getTag(thisRqst.Address)
17:         thisLine=cchL1[ndeIdx][numSet][lnIdx]
18:         if thisLine.TAG=thisTag then
19:             switch thisLine.State do
20:                 case MODIFIED
21:                     hitProcessing()                  ▷ REPLACEMENT POLICY update
22:                 case EXCLUSIVE || OWNED
23:                     hitProcessing()                  ▷ REPLACEMENT POLICY update
24:                     thisLine.State=MODIFIED
25:                     modifyLineInNxtLevel()
26:                 case SHARED || INVALID
27:                     LowPortsL1[ndeIdx]=thisRqst      ▷ ByPass request to nextLevel
28:                     thisLine.State=INVALID
29:             evaluateCoherenceProtocol()
30:             hitProcessing()                          ▷ REPLACEMENT POLICY update
31:         else
32:             LowPortsL1[ndeIdx]=thisRqst              ▷ ByPass request to nextLevel
```

Algorithm A.9 Requests in cache L2

```
1:  function CACHEL2_REQUEST(cchL2, LowPortsL2, UpPortsL2)
2:      ndeIdx = threadIdx.y
3:      thisRqst=UpPortsL2[ndeIdx]
```

```
4:    if thisRqst.RqstType=READ then
5:        lnIdx=threadIdx.x
6:        numSet=getNumSet(thisRqst.Address)
7:        thisTag=getTag(thisRqst.Address)
8:        thisLine=cchL2[ndeIdx][numSet][lnIdx]
9:        if thisLine.TAG=thisTag && thisLine.State!=INVALID then
10:           hitProcessing()                          ▷ REPLACEMENT POLICY update
11:       else
12:           LowPortsL2[ndeIdx]=thisRqst              ▷ ByPass request to nextLevel
13:   else if thisRqst.RqstType=WRITE then
14:       lnIdx=threadIdx.x
15:       numSet=getNumSet(thisRqst.Address)
16:       thisTag=getTag(thisRqst.Address)
17:       thisLine=cchMap[ndeIdx][numSet][lnIdx]
18:       if thisLine.TAG=thisTag then
19:           switch thisLine.State do
20:               case MODIFIED
21:                   hitProcessing()                  ▷ REPLACEMENT POLICY update
22:               case EXCLUSIVE || OWNED
23:                   hitProcessing()                  ▷ REPLACEMENT POLICY update
24:                   thisLine.State=MODIFIED
25:                   modifyLineInNxtLevel()
26:               case SHARED || INVALID
27:                   LowPortsL2[ndeIdx]=thisRqst      ▷ ByPass request to nextLevel
28:                   thisLine.State=INVALID
29:           evaluateCoherenceProtocol()
30:           hitProcessing()                          ▷ REPLACEMENT POLICY update
31:       else
32:           LowPortsL2[ndeIdx]=thisRqst              ▷ ByPass request to nextLevel
33:   else if thisRqst.RqstType=UPDATE_LINESTATE then
34:       lnIdx=threadIdx.x
35:       numSet=getNumSet(thisRqst.Address)
36:       thisTag=getTag(thisRqst.Address)
37:       thisLine=cchL2[ndeIdx][numSet][lnIdx]
38:       if thisLine.TAG=thisTag then
39:           thisLine.State=MODIFIED
```

Algorithm A.10 Requests in cache L3

```
1:  function CACHEL3_REQUEST(cchL3, LowPortsL3, UpPortsL3)
2:      ndeIdx = threadIdx.y
3:      thisRqst=UpPortsL3[ndeIdx]                   ▷ Only one port
4:      if thisRqst.RqstType=READ then
5:          lnIdx=threadIdx.x
6:          numSet=getNumSet(thisRqst.Address)
```

```
7:      thisTag=getTag(thisRqst.Address)
8:      thisLine=cchL3[ndeIdx][numSet][lnIdx]
9:      if thisLine.TAG=thisTag && thisLine.State!=INVALID then
10:         hitProcessing()                          ▷ REPLACEMENT POLICY update
11:         returnLine(EXCLUSIVE_state)
12:     else
13:         LowPortsL3[ndeIdx]=thisRqst              ▷ ByPass request to nextLevel
14:   else if thisRqst.RqstType=WRITE then
15:     lnIdx=threadIdx.x
16:     numSet=getNumSet(thisRqst.Address)
17:     thisTag=getTag(thisRqst.Address)
18:     thisLine=cchL3[ndeIdx][numSet][lnIdx]
19:     if thisLine.TAG=thisTag then
20:       switch thisLine.State do
21:         case EXCLUSIVE || MODIFIED
22:           thisLine.State=MODIFIED
23:           hitProcessing()                        ▷ REPLACEMENT POLICY update
24:           returnLine(MODIFIED_state)
25:     else
26:       LowPortsL3[ndeIdx]=thisRqst              ▷ ByPass request to nextLevel
27:   else if thisRqst.RqstType=WRITEBACK then
28:     lnIdx=threadIdx.x
29:     numSet=getNumSet(thisRqst.Address)
30:     thisTag=getTag(thisRqst.Address)
31:     thisLine=cchL3[ndeIdx][numSet][lnIdx]
32:     if thisLine.TAG=thisTag then
33:       thisLine.State=MODIFIED
```

Algorithm A.11 Responses in cache L1

```
1: function CACHEL1_RESPONSE(cchL1, LowPortsL1, UpPortsL1)
2:   ndeIdx = threadIdx.y
3:   thisRsps=LowPortsL1[ndeIdx]
4:   if thisRsps.RspType=(HTL2||HTSNP||HTL3||MSL3) then
5:     lnIdx=threadIdx.x
6:     numSet=getNumSet(thisRqst.Address)
7:     thisTag=getTag(thisRqst.Address)
8:     thisLine=cchMapL1[ndeIdx][numSet][lnIdx]
9:     if thisLine.State=INVALID || thisLine.usedBit=NO then
10:      insertLine(lnIdx)
11:    if anyLineIsINVALID() or anyLineIsNotUsed() then
12:      insertLine(randomIdx)
```

Algorithm A.12 Responses in cache L2

```
1: function CACHEL2_RESPONSE(cchL2, LowPortsL2, UpPortsL2)
```

```
2:    ndeIdx = threadIdx.y
3:    thisRsps=LowPortsL2[ndeIdx]
4:    if thisRsps.RspType=(HTSNP||HTL3||MSL3) then
5:        lnIdx=threadIdx.x
6:        numSet=getNumSet(thisRqst.Address)
7:        thisTag=getTag(thisRqst.Address)
8:        thisLine=cchL2[ndeIdx][numSet][lnIdx]
9:        if thisLine.State=INVALID || thisLine.usedBit=NO then
10:           insertLine(lnIdx)
11:       if anyLineIsINVALID() or anyLineIsNotUsed() then
12:           insertLine(randomIdx)
13:       if replacedLine.State==MODIFIED then
14:           writeBackRequestToNextLevel()
```

Algorithm A.13 Responses in cache L3

```
1:  function CACHEL3_RESPONSE(cchL3, LowPortsL3, UpPortsL3)
2:    ndeIdx = threadIdx.y
3:    thisRsps=LowPortsL3[ndeIdx]                          ▷ Only one port
4:    if thisRsps.RspType=(MSL3) then
5:        lnIdx=threadIdx.x
6:        numSet=getNumSet(thisRqst.Address)
7:        thisTag=getTag(thisRqst.Address)
8:        thisLine=cchL3[ndeIdx][numSet][lnIdx]
9:        if thisLine.State=INVALID || thisLine.usedBit=NO then
10:           insertLine(lnIdx)
11:       if anyLineIsINVALID() or anyLineIsNotUsed() then
12:           insertLine(randomIdx)
13:       if replacedLine.State==MODIFIED then
14:           writeBackRequestToNextLevel()
```

A.3 Shared bus Kernels

Algorithm A.14 Managing request from L2 caches

```
1: function BUS_ASSIGNER(L2Ports, SnpMsgPort, AdressesRegister)
2:     ndeIdx = threadIdx.x + blockDim.x * blockIdx.x;
3:     thisRqst=L2Ports[ndeIdx]
4:     if thisRqst.RqstType=(READ||WRITE||WRITEBACK) then
5:         exclusiveAssignment(thisRqst)
6:         snoopRequest(thisRqst,SnpMsgPort)
7:         registerLineAddress(AdressesRegister,thisRqst.Address)
```

Algorithm A.15 Snooping process

```
1: function SNOOP_PROCESS((SnpMsgPort, queueL3Port, cchL2, L2Ports)
2:     if SnpMsgPort.RqstType = (READ || WRITE) then
3:         ndeIdx = threadIdx.y
4:         lnIdx=threadIdx.x
5:         numSet=getNumSet(thisRqst.Address)
6:         thisTag=getTag(thisRqst.Address)
7:         thisLine=cchL2[ndeIdx][numSet][lnIdx]
8:         if ((thisLine.TAG=thisTag) &&
                   (thisLine.State=
                   (EXCLUSIVE||MODIFIED||OWNED))) then
9:             snoopHitProcessing(L2Ports)          ▷ Protocol coherence process
10:        if ndeIdx=0 && lnIdx=0 then              ▷ Only one cudaThread
11:            if NotSnoopHit() then
12:                missProcessing(queueL3Port)      ▷ ByPass request to nextLevel
```

Algorithm A.16 Access queue to L3

```
1: function QUEUETOL3(queueL3Port, dataCrclBffr, L3Ports)
2:     if queueL3Port.RqstType = (READ || WRITE || WRITEBACK) then
3:         if L3IsFree() then
4:             L3Ports=snoopRqst
5:         else
6:             addQueueAccessL3(snoopRqst,dataCrclBffr)
7:     else if L3IsFree() && dataInQueueAccessL3() then
8:         L3Ports=dataInQueueAccessL3(top)
```

Algorithm A.17 Response from L3

```
1: function RESPONSEFROML3(L3Ports, dataCrclBffr, L2Ports)
2:     if L3Ports.RqstType = (L3HIT || MEMHIT) then
3:         releaseL3(dataCrclBffr)
4:         L2Ports[L3Ports.RqstrNode]=L3Ports
```

Bibliography

[1] F. Accellerator. *Nallatech FPGA*. 20. URL: http://www.nallatech.com/ (visited on 01/01/2017).

[2] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. "Compute caches". In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2017, pp. 481–492.

[3] J. H. Ahn, S. Li, O Seongil, and N. P. Jouppi. "McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling". In: *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE. 2013, pp. 74–85.

[4] A. Akram and L. Sawalha. "x86 computer architecture simulators: A comparative study". In: *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 2016.

[5] A. Akram and L. Sawalha. "A survey of computer architecture simulation techniques and tools". In: *Ieee Access* 7 (2019), pp. 78120–78145.

[6] G. R. Andrews. *Foundations of multithreaded, parallel, and distributed programming*. Addison-Wesley, 2020.

[7] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. "COTSon: infrastructure for full system simulation". In: *ACM SIGOPS Operating Systems Review* 43.1 (2009), pp. 52–61.

[8] T. Austin, E. Larson, and D. Ernst. "SimpleScalar: An infrastructure for computer system modeling". In: *Computer* (2002).

[9] B Barnes and J Slice. "Simnow: A fast and functionally accurate amd x86-64 system simulator". In: *Tutorial at the IEEE International Workload Characterization Symposium*. 2005.

[10] D. Bartholomew. "Qemu: a multihost, multitarget emulator". In: *Linux Journal* 2006.145 (2006), p. 3.

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M.

Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: http://doi.acm.org/10.1145/2024716.2024718.

[12] R. E. Bryant, O. David Richard, and O. David Richard. *Computer systems: a programmer's perspective*. Vol. 2. Prentice Hall Upper Saddle River, 2003.

[13] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam. "The VPC trace-compression algorithms". In: *IEEE Transactions on Computers* (2005).

[14] A. Butko, R. Garibotti, L. Ost, V. Lapotre, A. Gamatie, G. Sassatelli, and C. Adeniyi-Jones. "A trace-driven approach for fast and accurate simulation of manycore architectures". In: *The 20th Asia and South Pacific Design Automation Conference*. IEEE. 2015, pp. 707–712.

[15] T. E. Carlson, W. Heirman, and L. Eeckhout. "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2011, p. 52.

[16] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. "An evaluation of high-level mechanistic core models". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 11.3 (2014), pp. 1–25.

[17] J. Ceng, W. Sheng, J. Castrillon, A. Stulova, R. Leupers, G. Ascheid, and H. Meyr. "A high-level virtual platform for early MPSoC software development". In: *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM. 2009.

[18] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP*. 2008.

[19] S. Charles, A. Ahmed, U. Y. Ogras, and P. Mishra. "Efficient cache reconfiguration using machine learning in NoC-based many-core CMPs". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 24.6 (2019), pp. 1–23.

[20] X. Chen, G. Zhang, H. Wang, R. Wu, P. Wu, and L. Zhang. "MRP: mix real cores and pseudo cores for FPGA-based chip-multiprocessor simulation". In: *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2015.

[21] H.-Y. Cheng, J. Zhan, J. Zhao, Y. Xie, J. Sampson, and M. J. Irwin. "Core vs. uncore: the heart of darkness". In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE. 2015.

[22] *ChipsAndCheese Measuring GPU Memory Latency*. [Online]. Accessed: 2023-01-10.

[23] J. Chromczak, M. Wheeler, C. Chiasson, D. How, M. Langhammer, T. Vanderhoek, G. Zgheib, and I. Ganusov. "Architectural enhancements in intel® agilex™ fpgas". In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2020, pp. 140–149.

[24] C. Collange, M. Daumas, D. Defour, and D. Parello. "Barra: A parallel functional simulator for gpgpu". In: *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE. 2010, pp. 351–360.

[25] J. Cong, Z. Fang, M. Gill, and G. Reinman. "PARADE: A Cycle-Accurate Full-System Simulation Platform for Accelerator-Rich Architectural Design and Exploration". In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press. 2015.

[26] T. M. Conte, M. A. Hirsch, and K. N. Menezes. "Reducing state loss for effective trace sampling of superscalar processors". In: *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*. IEEE. 1996, pp. 468–477.

[27] Z. Cui, Y. Liang, K. Rupnow, and D. Chen. "An accurate GPU performance model for effective control flow divergence optimization". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE. 2012, pp. 83–94.

[28] L. Duan and W.-M. Lin. "Hardware vs. Simulation: Bridging the Gap between Efficiency and Flexibility for Computer Engineering Education and Research". In: *Proceedings of the 2015 ASEE Gulf-Southwest Annual Conference* (2015).

[29] L. Eeckhout. "Computer architecture performance evaluation methods". In: *Synthesis Lectures on Computer Architecture* (2010).

[30] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, et al. "Asim: A performance model framework". In: *Computer* 35.2 (2002), pp. 68–76.

[31] M. J. Flynn. "Very high-speed computing systems". In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909.

[32] L. Gao, J. Huang, J. Ceng, R. Leupers, G. Ascheid, and H. Meyr. "TotalProf: a fast and accurate retargetable source code profiler". In: *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM. 2009.

[33] A. Gebregiorgis, H. A. Du Nguyen, J. Yu, R. Bishnoi, M. Taouil, F. Catthoor, and S. Hamdioui. "A survey on memory-centric computer architectures". In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 18.4 (2022), pp. 1–50.

[34] D. Genbrugge and L. Eeckhout. "Statistical simulation of chip multiprocessors running multi-program workloads". In: *Computer Design, 2007. ICCD 2007. 25th International Conference on*. IEEE. 2007, pp. 464–471.

[35] D. Genbrugge, L. Eeckhout, and K. De Bosschere. "Accurate memory data flow modeling in statistical simulation". In: *Proceedings of the 20th annual international conference on Supercomputing*. ACM. 2006, pp. 87–96.

[36] T. Grass, A. Rico, M. Casas, M. Moreto, and E. Ayguadé. "TaskPoint: Sampled simulation of task-based programs". In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2016.

[37] T. Grass, A. Rico, M. Casas, M. Moreto, and E. Ayguadé. "TaskPoint: Sampled Simulation of Task-Based Programs". In: (2011).

[38] Q. Guo, T. Chen, Y. Chen, and F. Franchetti. "Accelerating Architectural Simulation Via Statistical Techniques: A Survey". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016).

[39] M. S. Haque, J. Peddersen, and S. Parameswaran. "CIPARSim: Cache intersection property assisted rapid single-pass FIFO cache simulation technique". In: *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2011, pp. 126–133.

[40] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[41] R. Jagtap, S. Diestelhorst, A. Hansson, and M. Jung. "Exploring system performance using elastic traces: Fast, accurate and portable".

In: *Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), 2016 International Conference on*. IEEE. 2016.

[42] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. "High performance cache replacement using re-reference interval prediction (RRIP)". In: *ACM SIGARCH computer architecture news* 38.3 (2010), pp. 60–71.

[43] A. Joshi, J. Yi, R. H. Bell, L. Eeckhout, L. John, and D. Lilja. "Evaluating the efficacy of statistical simulation for design space exploration". In: *2006 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE. 2006, pp. 70–79.

[44] J. Y. Joshua, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, and J. E. Smith. "The future of simulation: A field of dreams". In: *Computer* (2006).

[45] S. Jourdan, T.-H. Hsing, J. Stark, and Y. N. Patt. "The effects of mispredicted-path execution on branch prediction structures". In: *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique*. IEEE. 1996, pp. 58–67.

[46] H. M. Kamali and S. Hessabi. "AdapNoC: A fast and flexible FPGA-based NoC simulator". In: *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*. IEEE. 2016, pp. 1–8.

[47] S.-h. Kang, D. Yoo, and S. Ha. "TQSIM: A fast cycle-approximate processor simulator based on QEMU". In: *Journal of Systems Architecture* 66 (2016), pp. 33–47.

[48] A. M. Kaushik, M. Hassan, and H. Patel. "Designing predictable cache coherence protocols for multi-core real-time systems". In: *IEEE Transactions on Computers* 70.12 (2020), pp. 2098–2111.

[49] G. Keramidas, N. Strikos, and S. Kaxiras. "Multicore cache simulations using heterogeneous computing on general purpose and graphics processors". In: *2011 14th Euromicro Conference on Digital System Design*. IEEE. 2011, pp. 270–273.

[50] N. Kumar and A. Sahu. "DDGSim: GPU based simulator for large multicore with bufferless NoC". In: *2014 Annual IEEE India Conference (INDICON)*. IEEE. 2014, pp. 1–6.

[51] A. Kupriyanov, F. Hannig, and J. Teich. "High-Speed Event-Driven RTL Compiled Simulation". In: *International Workshop on Embedded Computer Systems*. Springer. 2004.

[52] B. C. Lee and D. M. Brooks. "Accurate and efficient regression modeling for microarchitectural performance and power prediction". In: *ACM SIGOPS Operating Systems Review*. Vol. 40. 5. ACM. 2006, pp. 185–194.

[53] B. C. Lee, J. Collins, H. Wang, and D. Brooks. "CPR: Composable performance regression for scalable multiprocessor models". In: *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2008, pp. 270–281.

[54] K. Lee and S. Cho. "In-N-Out: reproducing out-of-order superscalar processor behavior from reduced in-order traces". In: *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*. IEEE. 2011, pp. 126–135.

[55] K. Lee, S. Evans, and S. Cho. "Accurately approximating superscalar processor performance from traces". In: *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 238–248.

[56] A. Li, S. L. Song, J. Chen, X. Liu, N. Tallent, and K. Barker. "Tartan: evaluating modern GPU interconnect via a multi-GPU benchmark suite". In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2018, pp. 191–202.

[57] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures". In: *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*. 2009, pp. 469–480.

[58] Z. Li, Y. Zhang, J. Wang, and J. Lai. "A survey of FPGA design for AI era". In: *Journal of Semiconductors* 41.2 (2020), p. 021402.

[59] G. H. Loh, S. Subramaniam, and Y. Xie. "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration". In: *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 53–64.

[60] G. López-Paradís, A. Armejach, and M. Moretó. "gem5+ rtl: A Framework to Enable RTL Models Inside a Full-System Simulator". In: *50th International Conference on Parallel Processing*. 2021, pp. 1–11.

[61] H. Lv, Y. Cheng, L. Bai, M. Chen, D. Fan, and N. Sun. "P-GAS: Parallelizing a cycle-accurate event-driven many-core processor simulator using parallel discrete event simulation". In: *Proceedings of the 2010 IEEE Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society. 2010.

[62] J. Ma, H. Wan, X. Gao, and X. Long. "Gpu-based time parallel cache simulator". In: *2010 IEEE Youth Conference on Information, Computing and Telecommunications*. IEEE. 2010, pp. 407–410.

[63] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. "Simics: A full system simulation platform". In: *Computer* 35.2 (2002), pp. 50–58.

[64] P. S. Magnusson, F. Larsson, A. Moestedt, B. Werner, J. Nilsson, P. Stenström, F. Lundholm, M. Karlsson, F. Dahlgren, and H. Grahn. "SimICS/Sun4m: A Virtual Workstation." In: *Usenix Annual Technical Conference*. 1998.

[65] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset". In: *ACM SIGARCH Computer Architecture News* 33.4 (2005), pp. 92–99.

[66] Ł. Maśko and M. Tudruj. "Parallel Event–Driven Simulation Based on Application Global State Monitoring". In: *International Conference on Parallel Processing and Applied Mathematics*. Springer. 2013.

[67] N. Matloff. "Programming on parallel machines". In: *University of California, Davis* 39319 (2011).

[68] C. J. Mauer, M. D. Hill, and D. A. Wood. "Full-system timing-first simulation". In: *ACM SIGMETRICS Performance Evaluation Review* 30.1 (2002), pp. 108–116.

[69] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. "Graphite: A distributed parallel simulator for multicores". In: *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE. 2010, pp. 1–12.

[70] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun. "A modern primer on processing in memory". In: *Emerging Computing: From Devices to Systems*. Springer, 2023, pp. 171–243.

[71] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood. "A primer on memory consistency and cache coherence". In: *Synthesis Lectures on Computer Architecture* 15.1 (2020), pp. 1–294.

[72] N. Nethercote and J. Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan notices*. Vol. 42. 6. ACM. 2007, pp. 89–100.

[73] S. Nilakantan, K. Sangaiah, A. More, G. Salvadory, B. Taskin, and M. Hempstead. "Synchrotrace: Synchronization-aware architecture-agnostic traces for light-weight multicore simulation". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2015, pp. 278–287.

[74] A. Nocua, F. Bruguier, G. Sassatelli, and A. Gamatie. "ElasticSimMATE: A fast and accurate gem5 trace-driven simulator for multicore systems". In: *2017 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. IEEE. 2017, pp. 1–8.

[75] R. Nozal, B. Perez, J. L. Bosque, and R. Beivide. "Load balancing in a heterogeneous world: CPU-Xeon Phi co-execution of data-parallel kernels". In: *The Journal of Supercomputing* 75.3 (2019), pp. 1123–1136.

[76] *NVIDIA CUDA C++ Programming Guide*. [Online]. Accessed: 2023-01-10.

[77] *NVIDIA CUDA Toolkit documentation*. [Online]. Accessed: 2021-10-20.

[78] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. "GPU computing". In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899.

[79] S. Pal, K. Kaszyk, S. Feng, B. Franke, M. Cole, M. O'Boyle, T. Mudge, and R. Dreslinski. "HetSim: Simulating Large-Scale Heterogeneous Systems using a Trace-driven, Synchronization and Dependency-Aware Framework". In: *IEEE International Symposium on Workload Characterization (IISWC) 2020, Virtual, October 27-29, 2020*. IEEE, 2020, pp. 13–24. DOI: 10.1109/IISWC50251.2020.00011.

[80] A. Patel, F. Afram, S. Chen, and K. Ghose. "MARSS: a full system simulator for multicore x86 CPUs". In: *Proceedings of the 48th Design Automation Conference*. ACM. 2011, pp. 1050–1055.

[81] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. "Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs". In: *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 2010, pp. 2–11.

[82] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer. "HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing". In: *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE. 2011, pp. 406–417.

[83] S. Raghav, A. Marongiu, C. Pinto, D. Atienza, M. Ruggiero, and L. Benini. "Full system simulation of many-core heterogeneous SoCs using GPU and QEMU semihosting". In: *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*. 2012, pp. 101–109.

[84] S. Raghav, M. Ruggiero, A. Marongiu, C. Pinto, D. Atienza, and L. Benini. "GPU acceleration for simulating massively parallel many-core platforms". In: *IEEE Transactions on Parallel and Distributed Systems* 26.5 (2014), pp. 1336–1349.

[85] S. Raghav, M. Ruggiero, A. Marongiu, C. Pinto, D. Atienza, and L. Benini. "Gpu acceleration for simulating massively parallel many-core platforms". In: *IEEE Transactions on Parallel and Distributed Systems* 26.5 (2015), pp. 1336–1349.

[86] G. Ridley and B. Forget. "A simple method for rejection sampling efficiency improvement on SIMT architectures". In: *Statistics and Computing* 31.3 (2021), pp. 1–11.

[87] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros. "Splash-3: A properly synchronized benchmark suite for contemporary research". In: *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*. IEEE. 2016.

[88] D. Sanchez and C. Kozyrakis. "ZSim: fast and accurate microarchitectural simulation of thousand-core systems". In: *ACM SIGARCH Computer Architecture News* 41.3 (2013), pp. 475–486.

[89] K. Sangaiah, M. Lui, R. Jagtap, S. Diestelhorst, S. Nilakantan, A. More, B. Taskin, and M. Hempstead. "SynchroTrace: Synchronization-aware architecture-agnostic traces for lightweight multicore simulation of CMP and HPC workloads". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 15.1 (2018), pp. 1–26.

[90] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter. "Tejas: A java based versatile micro-architectural simulator". In: *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2015 25th International Workshop on*. IEEE. 2015.

[91] J. Schneider, J. Peddersen, and S. Parameswaran. "MASH {fifo}: A hardware-based multiple cache simulator for rapid FIFO cache analysis". In: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE. 2014.

[92] R. Shi, S. Potluri, K. Hamidouche, J. Perkins, M. Li, D. Rossetti, and D. K. D. Panda. "Designing efficient small message transfer mechanism for inter-node MPI communication on InfiniBand GPU clusters". In: *2014 21st International Conference on High Performance Computing (HiPC)*. IEEE. 2014, pp. 1–10.

[93] N. N. Sirhan. "Multi-core processors: concepts and implementations". In: *Available at SSRN 3628131* (2020).

[94] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong. "AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 27.4 (2022), pp. 1–27.

[95] P. Srinuan, P. Sigdel, X. Yuan, L. Peng, P. Darby, C. Aucoin, and N.-F. Tzeng. "GPU-Assisted Memory Expansion". In: *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE. 2021, pp. 1–8.

[96] S. Van den Steen, S. Eyerman, S. De Pestel, M. Mechri, T. E. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout. "Analytical Processor Performance and Power Modeling Using Micro-Architecture Independent Characteristics". In: *IEEE Transactions on Computers* 65.12 (2016), pp. 3537–3551.

[97] K. Straube, H. Zhang, C. Nitta, M. Farrens, and V. Akella. "HySIM: Towards a Scalable, Accurate and Fast Simulator for Manycore Processors". In: (2013).

[98] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanovic. "RAMP gold: an FPGA-based architecture simulator for multiprocessors". In: *Design Automation Conference (DAC), 2010 47th ACM/IEEE*. IEEE. 2010, pp. 463–468.

[99] *The Open Virtual Platforms (OVP) OVPsim*. [Online]. Accessed: 2021-10-20.

[100] V. Tiwari. "AMD's comeback in the CPU market share with the launch of Ryzen". In: *Int. J. Res. Circuits Devices Syst* 3 (2022), pp. 17–23.

[101] L. Törnqvist, P. Vartia, and Y. O. Vartia. "How should relative changes be measured?" In: *The American Statistician* 39.1 (1985), pp. 43–46.

[102] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. "Multi2Sim: A simulation framework for CPU-GPU computing". In: *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2012, pp. 335–344.

[103] *VANDAL SYNCHROTRACE*. [Online]. Accessed: 2023-01-10.

[104] R. A. Velásquez, P. Michaud, and A. Seznec. "BADCO: Behavioral Application-Dependent Superscalar Core Models". In: *International Journal of Parallel Programming* 43.1 (2015), pp. 130–157.

[105] H. Wan, X. Gao, X. Long, and Z. Wang. "GCSim: A GPU-based trace-driven simulator for multi-level cache". In: *International Workshop on Advanced Parallel Processing Technologies*. Springer. 2009, pp. 177–190.

[106] J. H. Weinstock, R. Leupers, and G. Ascheid. "Accelerating MPSoC Simulation Using Parallel SystemC and Processor Sleep Models". In: *Proceedings of the 9th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. ACM. 2017.

[107] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. "SimFlex: statistical sampling of computer system simulation". In: *IEEE MICRO Special Issue on Computer Architecture Simulation and Modeling* 26.PARSA-ARTICLE-2007-001 (2006), pp. 19–31.

[108] H. S. Wilf. *Algorithms and complexity*. AK Peters/CRC Press, 2002.

[109] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 programs: Characterization and methodological considerations". In: *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*. IEEE. 1995.

[110] B. Xypolitidis, R. Shabani, S. V. Khandeparkar, Z. Ul-Abdin, T. Nordstr, et al. "Towards Architectural Design Space Exploration for Heterogeneous Manycores". In: *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. IEEE. 2016.

[111] M. T. Yourst. "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator". In: *2007 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE. 2007.

[112] W. Zhang, H. Wang, Y. Lu, H. Chen, and W. Zhao. "A Loosely-Coupled Full-System Multicore Simulation Framework". In: *IEEE Transactions on Parallel and Distributed Systems* (2016).

[113] G. Zheng, G. Kakulapati, and L. V. Kalé. "Bigsim: A parallel simulator for performance prediction of extremely large parallel machines". In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* IEEE. 2004, p. 78.