Towards the Simulation and Emulation of Large-Scale Hardware Designs

Guillem López Paradís

Thesis supervisor: Dr. Miquel Moretó (BSC, UPC)

Co-Supervisor: Dr. Adrià Armejach (BSC, UPC)

Specialization in High Performance Computing
Polytechnic University of Catalonia

This dissertation is submitted for the degree of
Master in Innovation and Research in Informatics

Barcelona School of Informatics October 29th, 2020

Per ensenyar-me a centrar-me i no conformar-me fins a arribar on vulgui.
Pel coneixement pel gust d'aprendre.
Gràcies Avi.

Acknowledgements

First of all, I would like to show special gratitude to my advisors: Miquel Moretó and Adrià Armejach, for their guidance and help in this work. I especially thank them for being patient with me during the last years.

Second, I would like to thank my colleagues from the group at BSC. Multiple discussions and long working days have passed, but also, the meetings outside of work have been joyful.

I would also like to express my gratitude to my loved friends, whether from the UPC, from climbing, or any other place. Thanks for helping me to have such a good time.

Last but not least, without my family, I can not consider being where I am today. Gràcies per fer-me com sóc i per estar al meu costat.

Finally, I would like to thank Carme for being there during the last years. Your unconditional support has help me to continue and pursue my dreams.

This work has been supported by the Spanish Government (Severo Ochoa grants SEV2015-0493), by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316-P), by the Generalitat de Catalunya (contract 2017-SGR-1328), by Mont-Blanc 2020 (GA 779877), by DRAC (SIFECAT: 001-P-001723), by EPI (GA 826647) and the European HiPEAC Network of Excellence.

Abstract

The heritage of Moore's *law* has converged in a heterogeneous processor with a many-core and different application- or domain-specific accelerators. Having also finished the benefits of Dennard scaling, we have ended up in chips with a large area that cannot be powered all at the same time but have space to improve the performance.

As a result, there are no more big performance gains from technology, and the most promising solutions are the creation of very smart designs of existing modules or exploring new specialized architectures. It is already a reality to see commercial products with many accelerators integrated on the System-On-Chip (SoC).

Therefore, future chips' perspective is to continue increasing the complexity and number of hardware modules added to the SoC. Consequently, the complexity to verify such systems has increased in the last decades and will increment in the near future. The latter has resulted in multiple proposals to speed-up the verification in both academia and industry. It also corresponds to the main focus of this thesis resulting in two different contributions.

In the first contribution, we explore a solution to emulate a big Network-On-Chip (NoC) in an emulation platform such as an FPGA or a hardware emulator. Emulating a NoC of 16 cores is unfeasible even in a hardware emulation platform depending on cores' size, which is pretty big. For this reason, we have exchanged the cores by a trace-based packet injector that mimics the behavior of an Out-of-Order (OoO) core running a benchmark. This contribution has materialized in the design of the trace specification and implementation of the trace generator in a full-system simulator: gem5. In addition, a preliminary study with a simple NoC has been done in order to validate the traces, with successful results.

In the second contribution, we have developed a tool to perform functional testing and early design exploration of Register-Transfer Level (RTL) models inside a full-system simulator: gem5. We enable early performance studies of RTL models in an environment that models an entire SoC able to boot Linux and run complex multi-threaded and multi-programmed workloads. The framework is open-source and unifies gem5 with a HDL simulator: Verilator. Finally, we have made an evaluation of two different cases: a functional debug of an in-house Performance Monitoring Unit (PMU); a design space exploration of the type of memory to use with a Machine Learning (ML) accelerator: NVIDIA Deep Learning Accelerator (NVDLA).

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Processor performance has been increasing at a lower pace in the last decade due to the end of the benefits provided by Moore's law [51] and Dennard scaling [29] in the last decade. These golden rules provided better performance for the same hardware design every two years. The industry was very successful at delivering these directives, until in early 2000s these golden *laws* started not to deliver the same benefits due to the power budget of a chip. Vendors reduced the transistor size every new generation for a similar area until they reached a point where the power density was unmanageable by regular cooling options.

The latter limitation is called power wall, as states that a given area of transistors can only dissipate a certain power. To overcome this limitation, designs can be better optimized to the power budget and for example, only power the parts of a chip that are running at a certain point. The latter is very common in current commercial products, and as a result, all the parts of the core cannot be powered at the same time due to power constraints.

As a logical result, core performance reached a plateau where the improvements were mostly given by its new architecture but without enormous improvements as before. The simple solution to continue improving at the same pace was the introduction of multiple cores in the same chip. This created the illusion of having more performance budget and again, being able to offer better performance for newer chips. However, it also created a whole new problem: the programmability wall.

As a consequence, to fully use the peak performance of a processor has become a tedious task. Now it is important to take into account all the architecture details: multiple cores, multiple threads per core, dynamic voltage and frequency scaling, etc. Also, in the parallel programming domain, you need to be cautious about where data is located, how to synchronize between the cores and how to divide the work among the cores. These questions are not trivial and multiple solutions have been explored in the last years.

Moreover, parallelism is not easy to apply to all scenarios, since some applications cannot benefit from it and the software languages have not been successful at offering an easy solution for every case. However, having more cores enables indubitably being able to execute more programs at the same time, hence at every new generation, vendors have offered processors with more cores.

Offering more cores has also been a logical reaction to the additional area available as the shrink of transistors has continued. Having more area has also allowed to include more hardware blocks in the same chip and even explore the inclusion of specialized hardware. This can be stated as a current and future trend, since specialization opens the door to have large performance and power improvements for very specific tasks.

This mentioned specialization has created very heterogeneous processors with a lot of hardware that before was placed off-chip due to area constraints. It is now common to see in commercial products: different types of cores, a Graphics Processing Unit (GPU) and various small specialized hardware modules all in the same System-On-a-Chip (SoC). The next logical step is to follow this direction and design large specialized accelerators for certain tasks that deliver good trade-offs.

There are some clear application domains that have been growing in attention in the last decade: Machine learning, Genomics and Big Data are some of the most promising. Clearly, Machine Learning (ML) can be considered the leader in this category, currently living its best time with a boom in start-ups and commercial products in the recent years. For example, startups like Graphcore [36] or Blaze [18] have obtained big investments; also, established companies like Google [43], Microsoft [23] and NVIDIA [72] have launched products specially designed for ML.

1.1 Motivation and Goals of the Thesis

Having reviewed the Computer Architecture panorama in the last decades, now it is important to explain the motivation and origin behind this work. On one hand, this new era of Computer Specialization suggests a golden era for Computer Architects [38] because the improvements in performance will come from clever designs and custom hardware. Following this new trend, there is a new boom of fabricating more chips. Since single performance of a core will not improve dramatically, making a chip to do a specific task is now profitable.

On the other hand, having multiple and heterogeneous hardware modules on the same SoC requires complex integration and verification processes to validate the interactions of all the elements at the level of the SoC which puts additional pressure on an already hard problem of

verifying hardware designs [31]. The logical question that arises is if new methods and tools are necessary to verify this new wave of large heterogeneous custom chips.

Currently, the process of fabricating a chip is quite tedious and long, usually with a fix timing and small flexibility. Adding to this, the tools to make ASICs are very expensive and not available to everyone; also, there are still various aspects in the process of creating the chip that are specific to the fundry with sensible information that cannot be released. To make the matter worse, nearly all the tools are not open-source and even the format to configure FPGA's of the Intel or Xilinx are private.

All of the latter clearly states that there is a clear need in improving the tools to be able to make and verify large-scale hardware designs. Having these tools will enable to verify the designs faster and better. This will help to reduce the multiple errors that can be found on the integration of the multiple modules that a SoC is composed or it can help to discover rare interactions that can be difficult to discover and cause fatal errors or huge bottlenecks.

1.2 Contributions of the Thesis

This thesis explores and solves two of the problems currently present in different steps in the hardware design cycle. First, we look at how to improve the debug and design of a realistic system composed of a NoC with up to 32 cores when facing the normal constraints an emulation platform has. Second, we explore a better solution to integrate RTL models inside full-system simulators in order to make initial performance studies, and also debug the module's functionalities. As a consequence, we open the door to make early design decisions when adding RTL models in a SoC with a full software stack.

1.2.1 Tracing an Out-Of-Order core to Design a Network-On-Chip

Nowadays, in a typical chip with ten's of cores, the standard of inter-communication is a NoC. This technology is mature with multiple lines of successful commercial products available in the market. However, in the last years, the idea of having in a single chip ten's of cores to even hundred's of cores has become a reality and working prototypes are already available. Designing and debugging such systems can become a challenging problem because you do not need to only verify the correct working of the nodes but also its interaction.

Following an easy and well-known technique called *divide and conquer*, one could debug such system in parts. On one hand, the different nodes are debugged and on the other hand, the NoC is also debugged. This method of debugging the parts in isolation has the disadvantage of not testing the integration of the whole design, which is especially critical for large systems.

A practical way of testing the interactions is doing it in small clusters that mimic the final bigger one. This set-up is more manageable can could fit in big FPGA's or emulation platforms. Furthermore, simulation can also be used to obtain a hint of the possible requirements of the system to the NoC. Hence, a mix of both methods is used, doing simulations of such a system first to obtain an idea of the expected interactions and later test the system on an emulation platform.

Inevitably, such a system targeting a full ASIC flow needs to be verified on emulation platforms or FPGA's at some point. It gives the most accurate numbers but requires the use of a tedious tool-flow. However, using an FPGA to emulate all the SoC or even a small NoC can be prohibitive or impossible due to the size of such systems. The latter could be the reason to only try small portions of the system at once.

Another solution to the last problem is abstracting a whole IP's such as a core, with an expected behavior of such IP. The idea is to make traces of real applications in simulation per core to later be able to mimic its behavior with a traffic injector that only replays these traces. The final goal is to emulate the largest NoC possible by abstracting the cores with simple trace-based traffic injectors to reduce the size occupied in the emulation platform and put more *cores*.

The generated trace needs to be able to replicate the regular behavior of a core executing the application with requests to memory, stalls and arithmetic execution. It also needs to occupy as less as possible to not have huge traces that difficult its usage. Finally, these traces will be used on the emulation platform to be able to debug and verify that the NoC satisfies the expected performance. Being able to put more *cores* on these emulation platforms allows a better environment to check the performance numbers.

Our first contribution defines and develops an infrastructure inside a full-system simulator to trace up to 32 OoO cores. These traces are then evaluated on a first simple NoC, to later be able to check them on a final demonstrator. The latter will have a real NoC that will target 32 cores composed of a trace mimicking a core and a simple RTL module to insert the trace events. The details of this contribution can be found in Chapter 4.

1.2.2 Integrating RTL Models inside Full-System Simulators

SoCs have become increasingly complex in the last years as previously stated. They now incorporate a large number of hardware blocks on their designs. These blocks can be from standard cores up to very specific domain-specific accelerators. The mentioned blocks are usually developed in isolation to be later integrated in a number of different SoCs with different characteristics as a *black box*. As a result, these hardware blocks are often tested alone, similarly as the NoC problem explained before.

Without a comprehensive view of the target platform the hardware IP is going to be placed, the testing can be insufficient. In addition, the current available tools to perform functional testing and design space exploration analysis of these hardware blocks are typically simulation-based. One drawback in these simulation frameworks, is the constraint to only use *testbenches* that feed the interfaces of the module in isolation.

As a consequence, they do not model all the potential interactions and restrictions that may arise when the hardware block is integrated into a complex SoC with a complete software stack. For this reason, we introduce our second contribution: the gem5+RTL framework. It is a flexible infrastructure that enables easy integration of existing RTL models with the popular full-system gem5 simulator [17].

The framework enables to perform functional testing and design space exploration studies of existing RTL models on a full-system environment. This environment models an entire SoC able to boot Linux and run complex multi-threaded and multi-programmed workloads. We have evaluated the tool with two use-cases that demonstrate the usability and the type of studies this tool enables. The details of this contribution can be found in Chapter 5.

1.3 Thesis Organization

This document is structured in the following way:

- **Chapter 1** is the introduction of the problems that this thesis addresses and briefly explains the contributions made.
- **Chapter 2** summarizes the related work of this thesis.
- **Chapter 3** gives the methodology used.
- **Chapter 4** explains in detail the first contribution about *tracing OoO cores to design a NoC*.
- **Chapter 5** explains in detail the second contribution about *integrating RTL models inside a full-system simulator*.
- **Chapter 6** gives some final conclusions and possible future work.
- **Appendix A** explains the origin of some statistics used in Chapter 4, Section 4.6.
- **Appendix B** adds additional graphics on the evaluation of gem5+RTL made in Chapter 5, Section 5.4.

Chapter 2

State of the Art

The growing complexity of the SoC's [22] is bringing a productivity crisis requiring much more effort to bring a product into a reality every new generation. One of the most well-known cases of product delays in the industry is the case of Intel, delaying the adoption of the latest node, 10nm and 7nm, for their latest CPU's various years due to technical problems [64, 42, 7, 32]. Adding to this, the verification team is growing in a tremendous trend since 2000 [31, 21], in part thanks to the complexity of the chip's nowadays that is multi-core, heterogeneous and with multiple accelerators.

This scenario will not diminish in the following years and some proposals have been made to cope with this issue. One of the most promising, is the adoption of agile methodologies created in the software world around the early 2000 [16]. The application of the *Agile Manifesto* in the software ecosystem was a response for the increasing software complexity, missing deadlines and to improve the traditional established methodology of development. The previous standard way was also called *waterfall* because it consisted on rigid tasks among different teams that went sequential and only at the end of all the tasks, teams were encouraged to talk and get feedback from the client.

Agile development created some debate in the beginning [39], but in a decade, was the standard among the software industry [13] obtaining successful results and improving productivity. The main reason was the breakout of rigid phases and big tasks in multiple smaller tasks, constant or regular feedback from the client and multiple mini-iterations of each phase instead of one big cascade iteration.

A similar revolution has started in the hardware ecosystem, due to the previous motives stated before, and as a response to the high customization that the chips require nowadays. Back in 2010 there was already some debate about how to address the complexity of SoC's [26] and O. Shacham et al. [61] stated that the way of designing hardware must change, introducing

the need of re-usability and the concept of templates, very common in the software industry. They made a point on the complexity of chips, the need for customization and to reduce costs.

Some years later, Yunsup et al. [46] proposed the Agile approach to build microprocessors. This is a logic proposal after many years of successful operation in software development. This *hardware manifesto* is a culmination of the insights gained building 11 chips in 5 years. The application of agile in hardware is based on making multiple working prototypes, using the full tool-flow producing tape-out-ready designs that iteratively add new features. This drastically reduces the cost of verification and validation, very suitable for small teams. Although its not fully established in the industry, some companies like SiFive [5] and Agile Analog [1] encourage its usage.

The introduction of a new methodology has not been the only response to the increasing complexity of chips. A myriad of new tools have been developed to improve the efficiency of building new hardware. There have been some proposals to enhance traditional Hardware Description Languages (HDL) and others ones that create new HDLs. For example, while MyHDL [28] and PyMTL [49] proposes new frameworks to create RTL using the higher abstraction language Python, Chisel [14] proposes to use Scala as the higher abstraction language and finally, Bluespec [52] is based on a traditional HDL such as Verilog with enhancements to improve its semantics. In the end, all of them generate VHDL or Verilog as the standard HDL for the Electronic Design Automation (EDA) tools nowadays.

Therefore, there is a clear need to have tools to quickly test the functionality of these hardware blocks on a realistic target system that includes all the agents that may impact their behavior in an SoC design.

Evaluation of RTL models can be done using different tools and at different abstraction levels. From flexible software models in C++ that do not offer cycle accuracy but can obtain reasonable performance estimations, to HDL simulators that model the behavior of a given RTL model, very useful to discover functional bugs, and up to full SoC emulation in a FPGA that can give a good understanding of the overall performance of the system. Each of these solutions can be used depending on the step of the hardware design cycle, and sometimes are done in parallel if the verification team is large enough.

All the mentioned approaches are useful offering advantages and disadvantages. Software models can be of big interest to easily obtain performance numbers with big flexibility but since they are not modeling RTL, the performance numbers need to be validated and taken cautiously. RTL models are inevitably needed to create the final ASIC or FPGA solution, with slow simulation times but very useful to find functional bugs. Finally, FPGA emulation is very time consuming but offers the best accuracy when compared to the others, and is a mandatory step in any design that targets an ASIC or a FPGA.

2.1 High-Level Simulators

Despite the fact that new methodologies and tools are becoming available and some of them propose to directly test the ideas in RTL and reduce the use of simulators, it is probably not suitable in all scenarios and requires more effort for the developers. As mentioned before, there are already proposals to bring closer the high-level languages such as Python or C++ and building hardware. However, there is still some debate and we are far from a comprehensive solution. Hence, the standard before writing RTL is testing the ideas in high-level simulators.

Software simulators offer great flexibility to model from small to full-system OS-capable SoCs. Written in high-level languages like C++, usually achieve simulation speeds of a few kilo instructions per second (KIPS). Numerous simulators have been proposed that provide different levels of accuracy and simulation speed. Some of the relevant ones are the following:

COTSon [10] is a full-system simulator decoupling functional and timing simulation. Functional simulation relies on just-in-time compilation of the simulated program. COTSon features several levels of detail and supports sampling. In addition to performance, ESESC [9] also includes models for power consumption and thermal behavior. ESESC is the first simulator applying time-based sampling to simulation of multi-threaded applications.

Simulators based on dynamic binary translation are notorious for their higher simulation speeds, at the expense of full-system support, and are usually restricted to the ISA they execute on. *Sniper* [20] belongs to this category and features a purely analytic CPU model. Instead of modelling micro-architectural structures within the CPU, it employs the mechanistic *Interval Simulation* model [33]. Zsim [59] also uses dynamic binary translation, but with a novel parallelization technique called bound-weave that enables simulations of hundreds of cores. In addition, it provides lightweight user-level virtualization to support certain complex workloads that are usually restricted to full-system simulators.

For large-scale multi-node simulations of thousands of cores, the level of abstraction needs to be raised to make simulations feasible. In this context, MUSA [37] presents an end-to-end methodology that combines different levels of abstraction. MUSA is able to model the communication network, micro-architectural details, and system software interactions. Simulations are based on multiple levels of tracing, which difficults the integration of additional hardware blocks.

Finally, gem5 [17] is one of the most well-known full-system simulators among computer architects. gem5 supports multiple ISA's such as Armv8, x86_64, and RISC-V; in-order and out-of-order core models; multi-level cache hierarchies and different memory technologies. It can emulate peripheral devices in order to support an Operating System (OS) like a Linux kernel and to run multi-threaded and multi-process applications. Although the purpose of gem5 is not to be cycle accurate, but to offer an environment with all the necessary pieces present

on a real SoC running on a full software stack, together with tools like McPAT [47] can be used to obtain performance and area estimations. In addition, it is open-source and has a large community with contributions from both academia and industry. Therefore, it is a good a good starting point for early design exploration before moving onto the HDL domain.

There have been proposals to extend gem5 to use it as a pre-HDL tool in order to enable co-design of accelerators while taking into account the dynamic interactions within a SoC design. This enables to design efficient and balanced accelerator micro-architectures.

In this category, gem5-Aladdin [63] and PARADE [27] propose frameworks that combine the full-system features of gem5 and accelerator models generated from C code. The former obtains these C models from a pre-RTL tool called Aladdin [62], which takes high-level language descriptions and data dependence graph representations of the accelerator as inputs, and the latter uses High-Level Synthesis (HLS) tools. These approaches go one step beyond gem5 models, but once the design of the target hardware block is polished, a HDL implementation of the design is still needed.

2.2 Tracing Memory Coherence Behavior

As we have seen before, there are some simulators that make use of traces to speed-up simulations, for example, MUSA is trace-based. A trace could be defined as a sequence of events ordered chronologically to maintain consistency and coherency. Usually, a trace contains events for instructions and for memory operations separated. In addition, you can also add extra information to, for example, be able to account for power events.

The goal of the trace is to mimic the behavior of the application running in a core in a series of events. As a result, the logical problem of traces is usually the space it occupies. Some optimizations are usually applied like the usage of the binary format instead of a human readable one, and/or the usage of compression techniques.

A quite interesting topic coming from the traces is the creation of synthetic ones. The whole process of generating a trace can be tedious and long. Hence, a logical solution is to capture the behavior of applications in statistical models to later generate traces. These models occupies much less than regular traces and are able to generate them dynamically.

Following this research line, SynFull [15] proposes to use Markov chains and clustering techniques to obtain a model of a real application. The resulted model occupies much less than regular traces and is quite accurate. Its main advantage is capturing the application and cache coherence behavior, which is something that synthetic traces/traffic do not usually do.

2.3 HDL Simulators

RTL simulation is a necessary step in the verification process. It is usually slow, but offers high levels of accuracy that are instrumental to detect functional bugs. Several HDL simulators exist some with commercial and others with free-to-use licenses.

On one hand, comming from the industry, there are existing commercial HDL simulators from Cadence [2], Synopsis [6] and Mentor Graphics [4] the most important ones. In exchange of an expensive license, they offer: good performance, good compatibility with the last standards of VHDL and Verilog/System Verilog, and finally, interesting options to connect them to other tools and languages such as System C and C++.

On the other hand, on the open-source side, Verilator [66], Icarus Verilog [71] and GHDL [34] are the most important and updated HDL simulators. While Icarus Verilog offers good support for Verilog-2005, GHDL translates VHDL to machine code offering a good performance. Finally, Verilator offers high speed by compiling synthesizable Verilog to multi-threaded C++/SystemC. In addition, it is being used in industry and academia environments, due to the fact that offers a good level of performance when compared to the commercial solutions. In fact, companies like Tesla have recently decided to use Verilator internally [8].

2.3.1 Bridge between High-Level and HDL Simulators

High-level simulators like gem5 have always been criticized for not offering cycle accuracy, even when using other tools such as McPAT, the numbers obtained are not fully accurate and are open to some error [19]. However, there are many proposals to try to bring accuracy to this high level simulators: for example to integrate Verilator with other simulators [69, 30, 48]. In PAAS [48], Verilator is interfaced with gem5 with the focus of simulating FPGA-based accelerators. Verilator and gem5 run independently and communicate thorough Inter Process Communication (IPC). In addition, Verilator has also been integrated with Multi2Sim [69, 30], again with a focus to simulate systems that integrate an FPGA with a multi-core system via bus-based architectures.

2.4 FPGA Prototyping

If the bridge between simulators is a logic response to trying to obtain the most accurate models from different tools, using FPGA's to accelerate simulation is another path worth exploring. FPGA's are essential for verification processes but also are very interesting to accelerate simulations and in the recent literature, there has been good progress on fast and general FPGA-based simulators [24, 45, 58, 68]. Some of the problems of these frameworks are

the requirement of substantial investment in specialized FPGA boards [70], and the difficulty to use them for architectural design exploration since they are, in general, very hard to modify.

In addition, they use complex FPGA tool-chains that depending on the size and complexity of the design, can take hours in compilation time. Moreover, these frameworks usually focus on specific subsystems, since simulating entire SoCs can be prohibitive or unfeasible without enormous emulation platforms. Their use is typically restricted to the last stages of the design cycle, once the RTL model is mature enough to not lose countless time on the FPGA's emulation.

Luckily, there is a recent solution to overcome the problem of making big investments in FPGA equipment: Firesim [44] proposes to take advantage of the EC2 F1 instances of Amazon Web Services (AWS) that have powerful Xilinx FPGA's [60] at a reasonable cost. Firesim is an open-source full-system simulator accelerated through FPGA's that can simulate from very small hardware IP's, up to a network of thousand complex multi-node systems.

However, modifying the simulator to incorporate new hardware blocks can be tedious and complicated, as it requires modifying the simulator's RTL code. For this reason, they also provide the possibility to add software models into the simulator, easing the overall integration while enabling hybrid simulations.

To overcome the size requirements of full modern SoC's that cannot be emulated in regular FPGA's, hardware emulators are the commercial solution. The most known are Cadence Palladium, Mentor Veloce, and Synopsys Zebu [41]. These platforms are too expensive for regular groups in academia and are typically only available on commercial companies.

To sum up, FPGA-accelerated solutions and hardware emulators are of great interest in the last stages of the design cycle for hardware components. However, these environments are less flexible than software solutions and require the whole design to be in RTL. Entire SoC simulations can be a challenging but necessary to run full complex workloads.

Chapter 3

Methodology

This section describes the configuration of the simulation infrastructure used for both works in this thesis: tracing applications to debug a NoC and the integration of RTL modules inside full-system simulators. Both efforts make use of the gem5 simulator which runs Ubuntu 16.04 with Linux kernel version 4.15. We simulate a contemporary SoC with up to 32 out-of-order cores and a detailed multi-level memory hierarchy.

3.1 Tracing an Out-Of-Order core to debug a Network-On-Chip

We have configured gem5 to match the architecture envisioned in Mont-Blanc 2020. The main objectives of the project consist in designing and delivering hardware IP blocks to contribute in the development of a competitive european processor. The target system is composed of up to 32 cores. Table 3.1 shows the architectural parameters we employ to collect the traces. The vector processing units can be configured to have different vector lengths following the Scalable Vector Extension (SVE) [67] specification: from 128 to 2048 bits.

During the different phases of the project, the number of cores used in the simulator has changed from 1 for debugging purposes to up to 32 to test the limits of some applications and the tracing infrastructure itself. The default core configuration has resulted in 16 cores which offers a good balance in trace size, simulation timing and application performance.

The number of cores and the SVE length are the two main parameters that change when tracing the applications. These parameters are given to gem5 through the Command Line Interface (CLI) and multiple scripts have been created to automatize the process of simulating an application, to check the output and to polish the final traces obtained.

Processor size	1 - 8 - 16 - 32 cores
Cores	3-wide issue/retire, 64-entry instruction queue, 192-entry ROB, 48 LDQ + 48 STQ, 2 vector processing units (VPU), 2GHz
Private Caches	L1I: 48KB, 3-way, 2 cycle, 2 ports, 8 MSHR LID: 32KB, 2-way, 2 cycle, 2 ports, 24 MSHR L2: 256KB, 4-way, 7 cycle, 24 MSHR
Last-level Cache	8MB, 16-way, 64B lines, 8 banks, 32 MSHR per bank Data bank access latency of 9 cycles.
NoC	Coherent crossbar, 128-bit wide, 2 cycles
Main Memory	4 DDR4-2400 channels, 2 ranks/channel, 16 banks/rank, 8KB row-buffer 128-entry write and 64-entry read buffers per channel 75GB/s peak bandwidth. Bank conflicts and queuing delays modeled

Table 3.1 Parameters for Tracing-applications full-system simulations.

3.2 Integrating RTL Models inside Full-System Simulators

The gem5+RTL has been configured to model an Arm-based SoC full-system environment that models the application, the operating system, and the architecture in detail. Table 3.2 details the architectural parameters. Note that, there are different memory technologies as part of the design space exploration study that is explained on Chapter 5.

Between the core configurations detailed in Tables 3.1 and 3.2, only minor differences exists due to going for a more aggressive core and the different memory technologies used for a design space exploration study. Another difference is the usage of only one core to perform the experiments. However, multiple instances of NVDLA accelerators attached to the core have been useful to create bottlenecks on the system. Finally, the frequency of the accelerators has also been modified (0.5 - 1 - 2 GHz).

It is worth mentioning that the memory configuration used in this work tries to mimic as much as possible the real configurations available in the market. Memory types like GDDR5 and HBM have pretty standard configurations, and only in the case of DDR4, the number of channels has been changed in the experiments to reflect different configurations found in the market. In the end, the number of memory channels determines the amount of memory bandwidth available.

Processor size	1 cores
Cores	3-wide issue/retire, 92-entry instruction queue, 192-entry ROB, 48 LDQ + 48 STQ, 2GHz
Private Caches	L1I: 64KB, 4-way, 2 cycle, 8 MSHR LID: 64KB, 4-way, 2 cycle, 24 MSHR L2: 256KB, 8-way, 9 cycle, 24 MSHR, stride prefetcher
Last-Level Cache	16MB, 16-way, 64B lines, 8 banks, 32 MSHR per bank Data bank access latency of 20 cycles.
NoC	Coherent crossbar, 128-bit wide, 2 cycles
Main Memory	**DDR4-2400:** 2 ranks per channel, 16 banks per rank 8KB row-buffer, 128-entry write, 64-entry read buffers per channel, 18.75GB/s peak bandwidth per channel **GDDR5:** quad-channel, 16 banks/channel, 2KB row-buffer 128-entry write and 64-entry read buffers per channel 112GB/s peak bandwidth **HBM:** 8 channels, 16 banks/channel, 2KB row-buffer 128-entry write, 64-entry read buffers per channel 128GB/s peak bandwidth
PMU	Configured with 20 32-bit counters
NVDLA	2048 8-bit MACs, 512 KiB buffer, 1GHz

Table 3.2 Parameters for gem5+RTL full-system simulations.

3.3 Real Machines Used

In order to make the experiments described in the following chapters, two types of machines have been used: the first being a laptop and the second a cluster located at the Computer Architecture Department (DAC) at UPC. The development and testing has been generally done in the laptop with the exception of some compilations of RTL which required up to 32GB of RAM, rendering the laptop useless for this task.

For the timing experiments, both the laptop and the cluster have been used, and different runs have been made in order to remove outliers. gem5 simulator is deterministic so its final result will be independent to the machine running it but its simulation speed may vary. It has

been found that in some timing experiments the variation was higher in the server than in the laptop although gaining the node exclusively for the task. For this reason, the majority of the timing experiments have been done in the laptop when possible.

The main configuration of the machines are:

- Laptop

  CPU: Intel® Core™ i7-7600U CPU @ 2.80GHz × 4 (2 physical cores)

  Main Memory: 16GBx1, DDR4, 2400 MT/s

  Memory: 500 GB SSD

- Server Sert (Latest node)

  CPU: AMD® EPYC™ 7401P 24-Cores @ 3GHz

  Main Memory: 128GB DDR4

  Memory: 1TB Sata-3 HDD

Chapter 4

Tracing an Out-Of-Order Core to Design a Network-On-Chip

The Mont-Blanc European project series started in 2011 with the ambition to research the usage of low-power embedded processor technology for High Performance Computing (HPC). After nearly a decade, the project has been divided in multiple phases, which has created and consolidated a viable software ecosystem for Arm in HPC. Mont-Blanc 2020 is the fourth phase of this ambitious project that concentrates its effort in designing and delivering hardware IP blocks to contribute to the development of a European processor for Big Data and HPC. The next phase is called European Processor Initiative (EPI) and is the culmination of the Mont-Blanc project series with a more ambitious plan, a competitive HPC processor.

One of the hardware IP blocks developed in Mont-Blanc 2020 is the NoC, which is the current standard of inter-communication in a chip with ten's of cores. For this reason, the project is expected to deliver at the end of 2021 a working NoC to be used on futures generations of European HPC processors. For the design of the NoC, it has been considered the requirements for HPC, Big Data and automotive sectors among others. To deliver such an IP, first an obvious analysis of the markets mentioned before is needed and has been done by some partners in the project. This analysis has delivered a list of real life applications that are important for these markets to be able to improve the performance of the final system.

Delivering a working NoC in RTL implies a huge effort in verification due to its criticality in the system. Moreover, emulating a NoC with ten's of cores (e.g. 32) is already a tedious task with RTL simulation and very expensive, or even impossible, in FPGA emulation of RTL models. Even if the Mont-Blanc 2020 consortium has access to one of the largest FPGAs available in the market, it has been decided to only emulate a NoC consisting of 8 or 16 cores with only its last-level cache (LLC). The core is not emulated but abstracted with trace-based traffic injector to mimic the behavior of a real application. The project is also expected to

deliver a final demonstrator using a powerful hardware emulator called Veloce to show the prototype working in order to make preliminary studies and validate the designed system.

This chapter describes the infrastructure employed to obtain application traces in order to feed the RTL packet injector, which is in charge of running the trace in the final demonstrator. Its goal is to abstract the complexity of modeling an out-of-order core in RTL by keeping only the essential signals and protocols to be able to work in cooperation with the other elements in the NoC. The injector will model three aspects of a core: traffic generation, responding to coherence traffic, and finally, abstract the power event handling.

Therefore, the aim is to generate application traces that capture application behavior by recording memory misses at the L1 cache and by encoding the memory and computing stalls the core is experiencing. By replaying these traces in the injector we can then generate realistic traffic into the NoC based on real applications. The trace is generated with the help of the gem5 simulator. Gem5 is configured to mimic the architecture envisioned in the Mont-Blanc 2020 project, with clusters up to 4 Armv8 cores that have private L1 and L2 caches, and a shared LLC across all clusters.

The following sections describe the trace specification, how we deal with certain corner cases and synchronization primitives, the different trace formats and their encoding, the steps we do to validate the generated traces, and finally a small example of an obtained trace in human readable mode.

4.1 Context

The work explained in this chapter corresponds to a task in the Mont-Blanc 2020 project that will be used in the final demonstrator. To put the task in some context, Figure 4.1 shows a simplified architecture similar to the one that will be used in the final demonstrator. The target architecture is composed of clusters of up to 4 Arm cores. Since emulating all the cores would be unfeasible, the decision is to model the core with an RTL packet injector that will read traces extracted from real applications. The modeled NoC in RTL includes a slice of the LLC and a home node (HN) for each core, that is, for each RTL packet injector.

Figure 4.1 shows a simple NoC with a ring topology and 4 clusters of cores. The proposed methodology is flexible and can be used with different topologies. As a result, the trace should be agnostic to the final topology to allow research in this direction. The figure also shows the example of a cluster inter-communicated and the final core emulation infrastructure which is the final context of the work in this chapter. Additionally, it can be seen on the right part of the figure, a zoom-in representing the exchange of a core by a portion of the LLC, packet injector and a trace, expressing that there should be a trace per core.

The packet injector is the final consumer of the trace and hence, multiple communications with the team developing it have been made. The final agreement is the trace specification that will be explained in detail in Section 4.2. The packet injector is a full hardware RTL IP developed in Verilog that reads the trace in binary format, creates memory commands to the LLC and also has the required logic to handle the trace events like waiting for some specific memory requests or waiting cycles when needed.

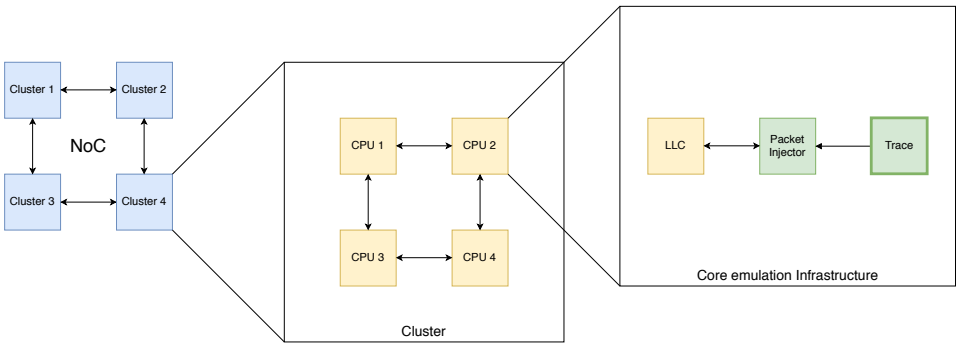


Fig. 4.1 Final Demonstrator System example.

Figure 4.2 shows the tracing infrastructure developed for the Mont-Blanc 2020 project. For each core being traced, there is a module that creates the trace file. This modules is located between the L1 and L2 caches. Also, for each core, a trace is created to ease the final demonstrator procedure.

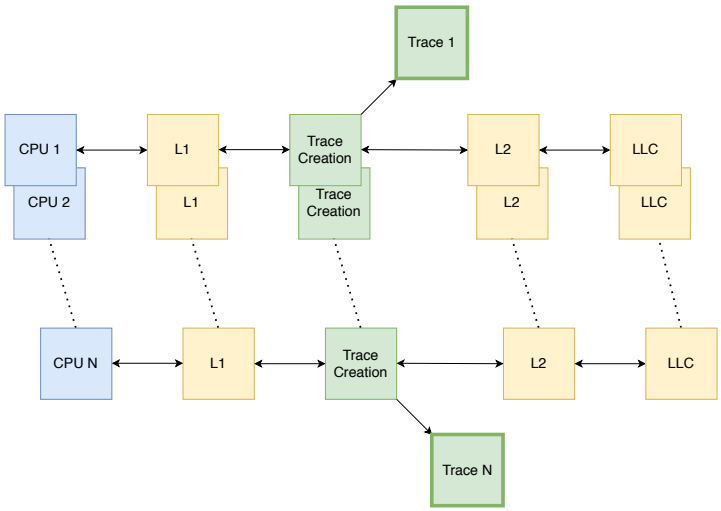


Fig. 4.2 Tracing infrastructure overview showing the exact place where the trace generation in the memory hierarchy is performed.

4.2 Trace Specification

4.2.1 Overview

The trace has to abstract the complexity of the underlying multi-core being simulated in gem5, while still capturing the behaviour of the applications being traced. Therefore, we want to capture the memory activity, such as: data access patterns, data locality, and contention; as well as the performance of the core, including compute stalls due to data dependencies and front-end stalls. The proposed trace format accomplishes this by using four different events:

- MISS: We identify L1 cache misses and tag them with an identifier, the instruction that triggered the miss, and the target physical memory address.
- WAIT_FOR: It is emitted when there is a memory related stall in the core. This happens when at the top of the reorder buffer (ROB) there is an instruction that has caused a miss (or a hit on miss) and it cannot be committed. This is a blocking event that indicates that the miss needs to be filled before the core can continue its execution. Similarly, the trace replay will have to stop untill the miss is filled.
- WAIT: Always present before a MISS event to indicate the number of cycles spent in compute since the previous MISS event occurred. This event takes into account internal pipeline stalls that occur in the different front-end stages such as fetch, decode, rename, and issue. The specified number of cycles does not include the cycles stalled due to a WAIT_FOR event.
- WFI/WFE: These two instructions specifies when the core goes to stand-by to wait for an interrupt or exception.

For each event we have already defined all the necessary fields. These allow to express all the necessary cases needed by the SVE RTL packet injector being developed by another partner in the project. Section 4.2.2 details all the fields each of the events has. In addition, we have defined the binary codification of the trace that the RTL injector assumes when reading the trace, see Section 4.4.

4.2.2 Event Description

In this section, the four events defined before are described in detail. The description of the events corresponds to the general form in human readable, and no optimizations done in the binary codification are described.

MISS

The MISS event occurs when there is an L1 cache miss. Table 4.1 lists the different fields a MISS event can have.

Position	Field	Description
1	Core id	Core identifier that creates this event
2	Event	Type of event
3	Req Id	Counter of the Req Id starting from 1 at the beginning of the trace
4	Address	Address in hexadecimal of the miss
5	D/I	Which cache creates the event, data or instruction
6	LD/ST	Whether the miss is from a load or a store
7	[E]	This states if it is an eviction
8	[X]	This states if the load or store is exclusive

Table 4.1 List of fields of a MISS event.

Table 4.2 shows a number of examples for MISS events. It takes into consideration loads, stores, evictions, and also exclusive loads and stores that are used in atomic constructs[1].

	Core id	Event	Req Id	Address	D/I L1	LD/ST	[E]	[X]
Load	0	MISS	1	0xfb697df0	D	LD		
Store	0	MISS	2	0xfb697df0	D	ST		
Eviction	0	MISS	3	0xfb697df0	D	ST	E	
LD/ST Exclusive	0	MISS	4	0xfb697df0	D	LD/ST		X

Table 4.2 MISS event examples.

WAIT_FOR

A WAIT_FOR event is generated when the top of the ROB is a load that has produced an L1 miss. This may cause a stall in the ROB if the load is not filled yet. As shown in Table 4.3, this event has only 2 fields: the physical address on which we have to wait for and the request id.

[1] https://developer.arm.com/documentation/dui0204/f/arm-and-thumb-instructions/memory-access-instructions/ldrex-and-strex

Position	Field	Description
1	Core id	Core identifier that creates this event
2	Event	Type of event
3	Address	Address in hexadecimal of the miss
3	Req Id	The Req Id of the miss

Table 4.3 List of fields of a WAIT_FOR event.

Table 4.4 shows an example for a WAIT_FOR event. When the packet injector reads the trace, it needs to stop until the memory operation described is satisfied.

Core id	Event	Req Id	Address
0	WAIT_FOR	2	0xfb697df0

Table 4.4 Example of WAIT_FOR event.

WAIT

The WAIT event gives information between two misses: it gives the number of cycles, committed instructions and committed SVE instructions between two misses. This event is emitted at the same time as the "new" miss and before emitting the actual MISS event.

Table 4.5 lists the different fields a WAIT event can have. The number of cycles between two misses (Position 3 in Table 4.5) is defined as the number of elapsed cycles minus the number of cycles stalled because the top of the ROB being a load that needs to be filled. These stalled cycles are captured by WAIT_FOR events. The number of WAIT_FOR events between two misses can be zero or more.

Position	Field	Description
1	Core id	Core id that creates this event
2	Event	Type of event
3	#Cycles	Number of cycles elapsed since previous MISS event
4	#Instructions	Number of committed instructions since previous MISS
5	#SVE Instructions	Number of committed SVE instructions since previous MISS

Table 4.5 List of fields of a WAIT event.

Table 4.6 shows an example for a WAIT event. The number of normal/SVE instructions give the required knowledge to the trace to later be able to check whether the given application trace has the expected memory and compute insensitivity.

Core id	Event	#Cycles	#Instructions	#SVE Instructions
0	WAIT	33	11	1

Table 4.6 Example of WAIT event.

WFI/WFX

The Wait For Interrupt (WFI) and Wait For Event (WFE) are two instructions for entering low-power standby state where most clocks are gated[2]. Table 4.7 lists the different fields a WAIT event can have. This is the last event to enter to the trace specification, currently not present on the traces due to probably being very "short". However, this event can be later injected artificially to also research on this direction on the NoC.

Position	Field	Description
1	CPU ID	CPU id of the CPU that creates this event
2	Event	Type of event

Table 4.7 List of fields of a WFI/WFE event.

Table 4.8 shows an example for a WFI and WFE event.

Core id	Event
0	WFI
0	WFE

Table 4.8 Example of WFI/WFE event.

[2]http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka15473.html

4.2.3 Implementation Details

The MISS event is emitted when there is an L1 cache miss. We check the miss status holding register (MSHR) and assert that it is an actual miss, i.e., there is no outstanding request for that address (e.g. hit on miss). At this point, we assign a new *request id* (Req Id) to the MISS event. Each simulated core owns and maintains this counter. In addition, there is a request to the core to find the instruction that generated the miss, and we associate the Req Id to this dynamic instruction. If the instruction generates a stall later, we are able to generate the WAIT_FOR event specifying the Req id of the offending instruction.

The WAIT_FOR event is emitted when at the top of the ROB there is an instruction that has caused a miss, which may cause a ROB stall if the miss has not been filled yet. As shown in Figure 4.3, when a load is at the top of the ROB we check whether it has a valid *request id* associated, if that is the case, then the WAIT_FOR event can be generated. Note that, while rare, there could be a stall in the ROB for accesses that actually hit in the L1, especially if the ROB has low occupancy, and in that case we do not have to emit a WAIT_FOR event. Similarly, a load that misses in the L1 data cache could still not have a valid Req Id associated if the actual miss has not happened yet (still accessing L1). In this case, we mark the instruction as pending and check at a later stage if it was an actual miss, as shown in Figure 4.3. Events of type WAIT_FOR are always emitted after the next WAIT event.

The WAIT event is always emitted just before a MISS event. It takes into account the number of instructions committed, as well as the number of cycles that have passed since the last MISS event. Note that the amount of cycles the core is stalled due to the ROB being blocked is not accounted for here, as this time is modeled by the events of type WAIT_FOR.

Figure 4.4 shows the main steps our tracing infrastructure does when detecting a MISS event. It first generates a new Req Id that is saved in the MSHR and the dynamic instruction objects. Then the event of type WAIT is generated and emitted. We check if there is a load at the top of the ROB marked as pending and emit all the WAIT_FOR events generated since the last MISS event. Finally, we emit the actual MISS event.

The following sections describe how we handle several critical scenarios in our traces. Including the mentioned stalls in the ROB, instruction cache stalls, multicore considerations, and atomic operations based on load-link store-conditional semantics.

Accounting for ROB Stall Time

Modelling ROB stalls due to memory latency is one of the key insights in our tracing methodology. The simple case illustrated in Figure 4.5 shows two MISS events separated by T cycles. It can be seen that between these two MISS events there could be a stall from the ROB during

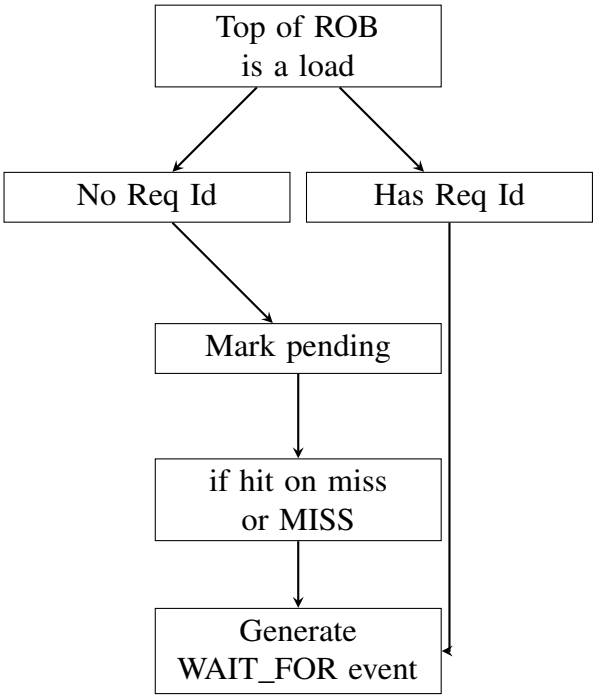


Fig. 4.3 Treatment of load instructions when present at the top of the ROB.

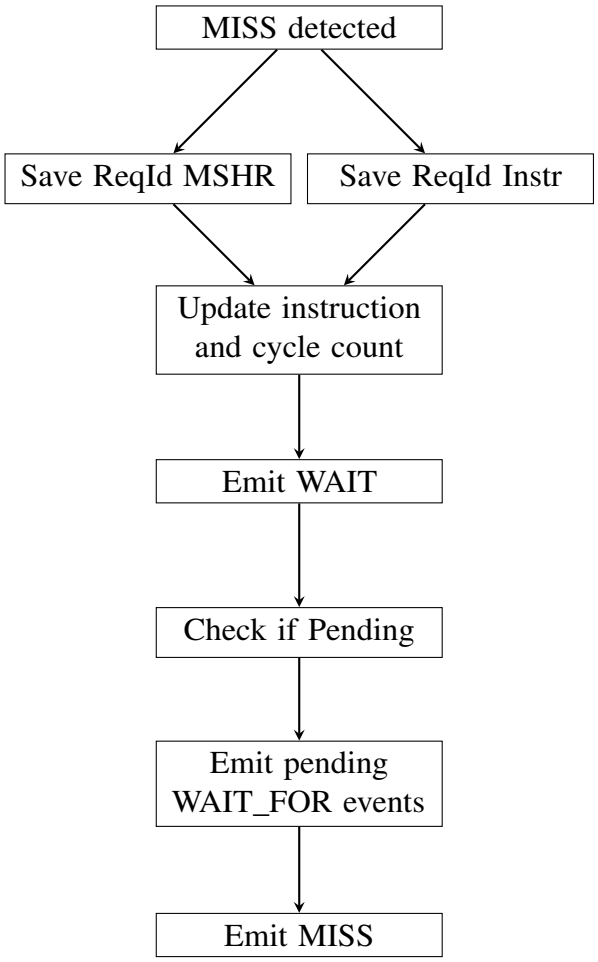


Fig. 4.4 Steps taken upon detecting a new MISS event.

T_{STALL} cycles. This means that during this period of time we cannot commit any instructions until the load is filled and the ROB is unstalled.

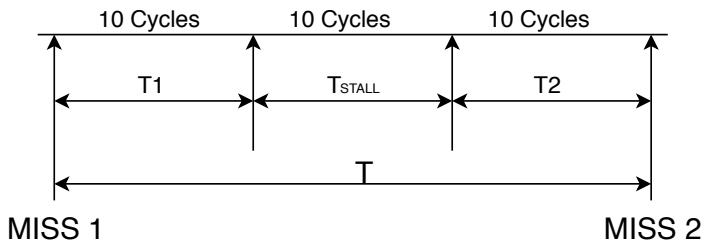


Fig. 4.5 Simple ROB stall.

To model this simple case we need to identify the minimum number of compute cycles between the two MISS events and also insert a blocking event (WAIT_FOR) that models the

stall in the ROB. Therefore, we want to have a WAIT event of $(T - T_{STALL})$ or $(T1 + T2)$ cycles, which is the minimum time to wait between these two misses. In addition to that, we will also generate a WAIT_FOR event for the address and request id of the load instruction that stalled the ROB. Figure 4.6 shows the corresponding trace for this simple example, where after the first MISS event we have to WAIT for a minimum number of cycles (computation with forward progress), and then WAIT_FOR a memory access to be filled by the memory hierarchy.

MISS	1		
(stall)	*not shown*		
(unstall)	*generate*	*wait_for*	*event*
WAIT	$T1 + T2$		
WAIT_FOR	1		
MISS	2		

Fig. 4.6 Simple ROB stall trace.

Although this is the general case, it can happen that while the ROB is stalled, there are new misses issued into the memory hierarchy. This makes the management of the stall time a little bit more complex. Figure 4.7 shows this scenario. If there are MISS events while on a ROB stall, the latency cost of these misses is free from the point of view of the core, since it is already waiting on another miss. These miss events that occur during the ROB stall need to be placed in the trace before we actually put the blocking WAIT_FOR event that triggered the current ROB stall. When the trace is replayed these MISS events can be treated before actually blocking the replay to honor the WAIT_FOR event.

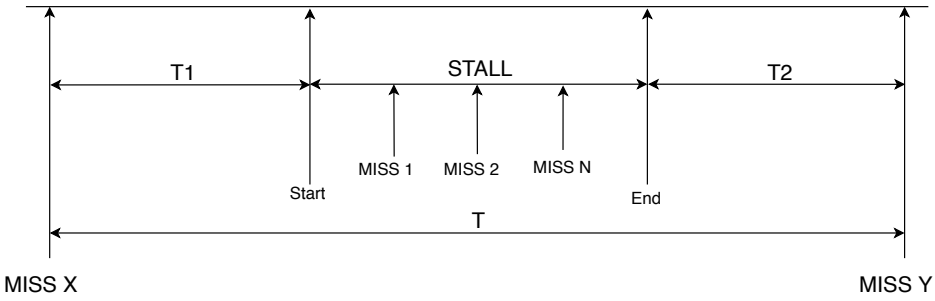


Fig. 4.7 Complex ROB stall.

Figure 4.8 shows how we construct the trace as events occur. A few considerations:

- MISS X and Y are misses outside the ROB stall.

- MISS 1,2 and N are the misses inside the ROB stall.
- The WAIT that needs to be emitted before MISS 1 is WAIT T1, so that it does not reflect the time between the start of the stall and MISS 1.
- After resuming from the stall, the WAIT before MISS Y would be WAIT T2, so that it does not reflect the time between the last MISS and the end of the stall.

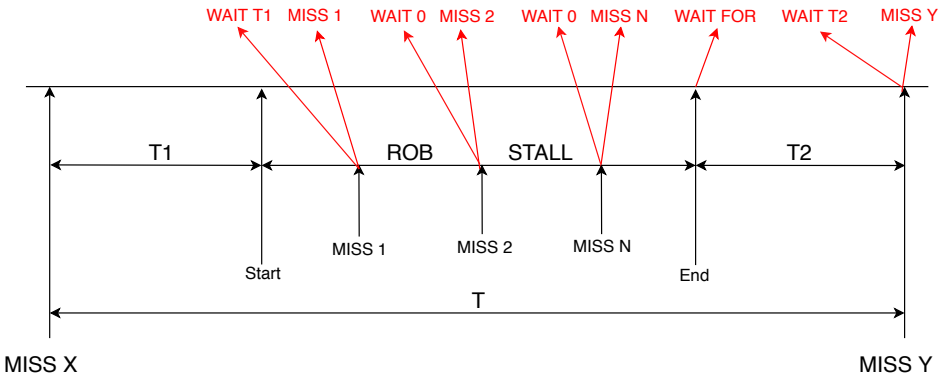


Fig. 4.8 Complex ROB stall with annotated events.

Figure 4.9 shows the resulting output trace.

MISS	X		
(stall)	*not shown*		
WAIT	**T1**		
MISS	1		
WAIT	**0**		
MISS	2		
WAIT	**0**		
MISS	N		
(unstall)	*generate*	*wait_for*	*event*
WAIT	**T2**		
WAIT_FOR	X		
MISS	Y		

Fig. 4.9 Complex ROB stall trace.

Reorder and Repeated WAIT_FOR Events

As mentioned earlier, we generate WAIT_FOR events for ROB stalls that may happen between two MISS events. These WAIT_FOR events are always emitted in the trace after the WAIT

event that indicated the minimum number of cycles that the core spent busy between two MISS events. In addition, we also filter possible repetitions of WAIT_FOR events over the same request id. These considerations can greatly simplify the logic of the packet injector that uses the trace as input.

Model Instruction Cache Stalls

Stalls produced due to instruction cache misses happen at the front-end of the pipeline. These stalls are already captured by the cycles we encode in the WAIT events. We do include the MISS events due to instruction cache misses, but the performance penalty associated to these misses is abstracted as part of the minimum amount of cycles that need to elapse between two MISS events.

Multicore Considerations

To ease the implementation of the packet injector that consumes the trace, the trace generator creates a trace file per core of the simulated multicore. This allows each packet injector instance to read from a different file sequentially, where all the events of the file are relevant.

Load-Link Store-Conditional Considerations

In order to support atomic operations, we have to support load-link (LL) and store-conditional (SC) semantics in the trace. When a core performs a load-link over a memory address, the subsequent store-conditional operation will only succeed if no other core has tried to write to that memory address. Otherwise, the store-conditional fails and the program counter is rolled back to re-execute the load-link again.

As agreed with the group implementing the packet injector, we only include in the trace LL and SC pairs that have executed successfully. This enables to model faithfully the behavior of LL/SC operations using the packet injector, as the outcome of these operations is highly dependent on memory and interconnection network latencies. Therefore, the packet injector is now in charge of handling failed LL/SC and re-execute them.

This implies that the trace generator needs to detect when a store-conditional fails and avoid emitting the LL/SC pair of events. This is achieved by marking events as pending after a LL until the SC succeeds, and not emitting any pending events. To avoid frequent input/ouput to the trace file, we employ a certain degree of buffering using an event list for the trace. This allows us to remove the events marked as pending when the SC fails. Note that MISS events that happen between a LL and a SC should not be marked as pending, as they have already been issued and will access the memory hierarchy.

Another caveat with LL/SC operations is that we include them in the trace even if it is an L1 cache hit. As we found cases where, for example, the LL is an L1 miss but the SC is a hit, which makes the trace hard to replay since only one of the two operations is present.

4.3 Methodology

As explained in Chapter 3, the implementation of the trace generator has been prototyped in gem5, including the definition of all the fields for each event and the ability to output a human readable and a binary trace format. There is a summary of the trace format in Section 4.4.

To obtain the traces, we have configured gem5 to match the architecture envisioned in Mont-Blanc 2020. Traces are meant to be agnostic with respect to the simulated memory hierarchy, as they abstract away the core and L1 caches, but are not influenced by the rest of the hierarchy, including the simulated NoC. Table 3.1 in Chapter 3 shows the architectural parameters we employ to collect the traces. The vector processing units can be configured to have different vector widths as per the SVE ISA specification, from 128 to 2048 bits.

We have modified different pipeline stages of the out-of-order core model, as well as the L1 caches, in the gem5 simulator. By doing so, we have been able to track miss request identifiers for dynamic instructions in order to generate all the necessary events and their fields.

Table 4.9 lists the Mont-Blanc 2020 applications, selected by another team inside BSC. We have been able to trace all applications except Arbor, due to an elaborated compilation procedure that we have not been able to complete successfully. under our gem5 infrastructure.

Application	**HPC dwarf**	**Prog. language**	**Prog. model**	**Vectorization**
Arbor	dense linear	C++	MPI + threading	Intrinsics
HACCKernels	n-body methods	C++	OpenMP	Auto-vectorization
HPCG	sparse linear	C++	OpenMP	Arm Perf. Libs
KKRnano	dense linear	Fortran	MPI + OpenMP	Intrinsics
Grid	structured grids	C++	MPI + OpenMP	Intrinsics
MiniAMR	structured grids	C	MPI	Assembly
RAJAPerf	set of HPC kernels	C++	OpenMP	Auto-vectorization
SWFFT	spectral methods	C++ and Fortran	MPI + OpenMP	Arm Perf. Libs
XSBench	embarassingly parallel	C	OpenMP	Auto-vectorization

Table 4.9 MB2020 applications. Indicating their computational dwarf, programming language, programming model, and SVE vectorization method.

Table 4.10 details the inputs employed for each benchmark when generating the traces.

Benchmark	Input
HACCKernels	1280
HPCG	24 24 24
KKRnano	poisson problem 1 3
Grid	–shm 1 –grid 4.4.4.4
MiniAMR	–num_refine 1 –max_blocks 18 –init_x 1 –init_y 1 –init_z 1 –npx 2 –npy 2 –npz 2 –nx 64 –ny 64 –nz 64 –num_objects 1 –object 2 0 -1.10 -1.10 -1.10 0.030 0.030 0.030 1.5 1.5 1.5 0.0 0.0 0.0 –num_tsteps 10 –checksum_freq 0 –stages_per_ts 10 –num_vars 1
EOS (RAJAPerf)	-v Base_OpenMP -k Lcals_EOS –repfact 0.0005 –sizefact 500
HYDRO (RAJAPerf)	-v Base_OpenMP -k Lcals_HYDRO_1D –repfact 0.001 –sizefact 250
LTIMES (RAJAPerf)	-v Base_OpenMP -k Apps_LTIMES –repfact 0.02 –sizefact 30
Stream-DOT (RAJAPerf)	-v Base_OpenMP -k Stream_DOT –repfact 0.002 –sizefact 100
VOL3D (RAJAPerf)	-v Base_OpenMP -k Apps_VOL3D –repfact .005 –sizefact 5
SWFFT	1 256
XSBench	-s small -p 80000

Table 4.10 Benchmark inputs used for tracing.

4.4 Trace Formats

In this section, it is explained how the trace is formatted. The trace can be generated in a human readable and/or binary formats to allow debugging but also the performance needed by the SVE RTL injector which reads 64 bits every time.

4.4.1 Human Readable Format

Table 4.11 summarizes the human readable format for each of the four events.

Some considerations:

- Event Id - is the event identifier, starting with 0.
- Req Id - is a counter attached to the miss events, starting with 1.
- Req Id with 0 is created artificially for the LL/SC operations on L1 hit.

Event Id	Core id	MISS	Req Id	@	D/I	LD/ST	E	X
Event Id	Core id	WAIT	#Cycles	#Instructions				
Event Id	Core id	WAIT_FOR	Req Id	@				
Event Id	Core id	WFI/WFE						

Table 4.11 Human readable format for each event.

4.4.2 Binary Format

Table 4.12 details the binary format that has been defined in agreement with the partner developing the RTL injector. Some considerations:

- Req Id: 22 bits are enough to cover all cases.
- Address: Physical addresses in gem5 are 40 bits. Since we use cache-line addresses we need 34 bits.

Type	ReqId[63:42]	Cycles/@[41:8]	OpC[7:5]	Excl[4]	Ld/St[3]	OpCode[2:0]
L1MissData	VVVVVV	Address	XXX	1	1	000
L1MissInst	VVVVVV	Address	XXX	0	0	001
EvictDirty	VVVVVV	Address	XXX	X	X	010
Wait For	VVVVVV	Address	XXX	X	X	100
Wait	Instr and Sve	Cycles	XXX	X	X	101

Table 4.12 Binary format for different events.

Type	[63:54]	[53:42]	Cycles[41:8]	OpC[7:5]	Excl[4]	Ld/St[3]	OpCode[2:0]
Wait	Sve Instr	Instr	Cycles	XXX	X	X	101

Table 4.13 Detailed binary format for WAIT event.

All events are encoded in 64 bits, which simplifies reading and processing the trace. Notes:

- D/I: 0 means data, 1 means instruction
- LD/ST: 0 means LD, 1 means ST
- Excl: 1 means exclusive (LL/SC)

4.5 Trace Validation

To debug and test our tracing infrastructure, we have performed a series of validation tests with simple micro-benchmarks. It has enabled us to verify that the output of the traces is aligned in our expectations in terms of the MISS events as well as compute and memory stalls. For example, we have created a simple vector addition program with configurable sizes which can be seen on the next page in the Listing 1. The code shows a simplified version of the real one used in the validation phase, to only exhibit the relevant parts like the subtract array function which is the vector addition loop. It is also present the necessary hooks in C that expresses the begin and end of the tracing of a given application.

In order to validate the tracing infrastructure we have checked statically the number of instructions in the vectorAddition code and later on, by disabling prefetchers and setting the size of the caches, we have determined the number of events that need to happen in terms of misses, evictions, etc. We then compare the results of these simulations with gem5 statistics and the trace output. The results have been very important to fix several bugs that have appeared during the development of the infrastructure.

The in-depth validation of the infrastructure with this specific test was continuously done until the trace specification was finished in order to check that any incremental improvement in gem5 didn't break anything. Afterwards, the obtained traces are only checked with a post-processing script. Having this highly configurability test helped in the debug of the packet injector HW IP because allowed to make very small traces of tens of events, and also with only specific events e.g. only misses of DCache.

Finally, once we obtain the traces from the simulation infrastructure, we also perform a set of syntactic validation checks to ensure the trace complies with certain formatting restrictions. For example, that all WAIT_FOR events have a matching MISS event. Otherwise, this could lead to a deadlock situation when replaying the trace if this condition is not met. These checks are performed automatically by a Python script that is capable to also get some statistics of the trace like the number of events or the total amount of wait cycles. This statistics are explained in depths in Section 4.6.

```c
#include <stdlib.h>
#include <stdio.h>

#define CACHESIZE 64*1024*8 // cache size example

void subtract_arrays(int *a, int *b, int *c, int size) {

        for (int i = 0; i < size; i+=1) {
                a[i] = b[i] - c[i];
        }
}

int main(int argc, char *argv[]) {

        int size = 0;
        int *a,*b,*c,*d;

        size = atoi(argv[1]);

        a = aligned_alloc(sizeof(int)*16,sizeof(int)*size);
        b = aligned_alloc(sizeof(int)*16,sizeof(int)*size);
        c = aligned_alloc(sizeof(int)*16,sizeof(int)*size);
        d = aligned_alloc(sizeof(int)*16,sizeof(int)*CACHESIZE);

        // Code to check if the mallocs were executed succesfully
        ...

        // Code to init the arrays
        ...

        // Hook that triggers the start of the the trace
        __parsec_roi_begin();

        subtract_arrays(a, b, c, size);

        // Hook that triggers the end of the trace
        __parsec_roi_end();

        return 0;
}
```

Listing 1 : Vector addition code used for the validation of the traces infrastructure.

4.6 Intial Evaluation: Trace Statistics

Having created the traces and validated them with the help of automated scripts, the next logical step is to verify if the traces performed well in a real scenario. Since the demonstrator platform is still under development and until beginning of 2021 is not planned to be available to use it, we had to change the scenario or stall this task. Luckily, there was a solution and we have been able to perform this task and make a preliminary study.

We have used an open-source NoC project developed in RTL [50] to perform early tests of the traces. Although both platforms do not have the same architecture, we can take this initial study as a hint of the final demonstrator performance. The task of executing the traces on this platform has been done by other researchers at BSC. The author of this thesis has done the analysis of the data gathered with this simulation infrastructure.

The evaluation platform consists of the NoC with the trace injector as cores without a main memory. Instead, the memory latency is fixed to a a reasonable average value. this set-up has also been useful to find bugs on the trace injector hardware module developed in Mont-Blanc 2020. Due to constraints in simulation time, we have simulated up to 8 core clusters in the NoC. Next, we have used some of the benchmarks traced and performed the experiments.

The goal of these experiments is to verify if the traces perform in a realistic environment and we are able to check if the application's performance obtained in gem5 can be replicated. It is important to mention than the packet injector only waits a maximum of 16 cycles on the *wait events*. The reason behind the latter is to reduce the total number of cycles waited on simulations and hence the total maximum time.

Table 4.14 shows the data obtained from the different experiments. We can highlight:

- Only a subset of benchmarks finished successfully due to some errors found on the trace injector side.
- Column *VL* corresponds to the Vector Length (VL) used in the simulations. VL 0 means scalar, VL 1 means 128 bits, VL 2 means 256 bits, VL 4 means 512 bits, VL 8 means 1024 bits and VL 16 means 2048 bits.
- Column *Wait Cycles* corresponds to the total Wait cycles of a given trace for all the cores.
- Column *gem5 Sim Cycles* is the total number of cycles simulated by gem5 when tracing the given benchmark for a determined VL.
- Column *Norm. NoC Cycles* is the total number of cycles that the NoC needed to run a given configuration of benchmark-VL. It is adjusted to not be affected by the maximum

number of 16 cycles for the events Wait imposed by the packet injector. A more detailed explanation this number can be obtained in Table A.2 in Appendix A.

- Column *Ratio* corresponds to the cycles obtained from the experiment (Norm. NoC) divided by the ones obtained from gem5.

Observing the results in detail we can see that for example in benchmarks like *Grid+Wilson* and *HACCKernels* the obtained simulated cycles and the ones from the experiment match and there is little difference to explain. However, in the other benchmarks, the average difference is between a 20% to 30% with some with only one case above 50% differences (EOS scalar reaches 94% difference).

The main reason that explains the big differences is due to the detail in the memory operations. For example, the *Wait For* events can take longer depending on the NoC implementation details and this will affect applications that are more memory bounded. Also, waiting a maximum of 16 cycles for any *wait event* can affect the final cycles since applications that have more cycles in wait events will now wait less and make possible stalls or congestions.

Finally, we can say we are happy for these preliminary results because for the first time we have been able to see the full tool-flow working. The traces work correctly and the simulation infrastructure enables the generation of additional traces on demand once the final demonstrator is working. Also, these experiments have been useful to find rare bugs in the packet injector.

APP NAME	VL	Wait Cycles	gem5 Sim Cycles	Norm. NoC Cycles	Ratio
EOS	0	109,678,673	117,657,163	228,491,578	**1.942011622**
EOS	1	115,504,138	115,608,281	166,037,773	**1.436210032**
EOS	2	115,521,478	115,556,871	153,164,460	**1.325446585**
EOS	4	93,433,615	93,456,808	111,615,601	**1.194301447**
EOS	8	100,453,506	105,814,109	139,382,534	**1.317239594**
Grid+Wilson	1	200,368,504	200,370,130	205,030,041	**1.023256515**
Grid+Wilson	8	104,509,940	104,513,878	108,086,201	**1.034180370**
HACCKernels	2	168,635,964	169,300,391	170,120,735	**1.004845494**
HACCKernels	4	129,875,129	129,947,090	130,882,217	**1.007196213**
HACCKernels	8	53,041,576	53,083,603	53,751,572	**1.012583340**
HPCG	2	245,896,744	245,989,448	306,232,992	**1.244902960**
LTIMES	0	249,367,520	249,370,459	269,473,024	**1.080613257**
LTIMES	2	158,525,845	158,528,769	193,133,071	**1.218284052**
LTIMES	4	124,750,389	124,753,718	157,369,133	**1.261438421**
LTIMES	8	72,145,613	72,147,566	96,288,191	**1.334600685**

Table 4.14 Statistics obtained from the execution of the traces in gem5 and the NoC.

4.7 Trace Example

Figure 4.10 shows a trace example in human readable format. This is a trace fragment from the simple vector addition validation benchmark obtained from one of the cores traced.

0 0 MISS	1	0xfb697df0	D	LD	
1 0 WAIT_FOR	1	0xfb697df0			
2 0 WAIT	33	11	2		
3 0 MISS	2	0x812cbbe0	I	LD	
4 0 WAIT	16	0	0		
5 0 MISS	3	0x812cbc00	I	LD	
6 0 WAIT	37	0	0		
7 0 MISS	4	0xfb5b2060	D	LD	
8 0 WAIT	1	0	0		
9 0 WAIT_FOR	4	0xfb5b2060			
10 0 MISS	5	0xfb5b20b0	D	LD	
11 0 WAIT	15	0	1		
12 0 MISS	6	0xfc01a080	D	ST	E
13 0 WAIT	7	10	1		
14 0 WAIT_FOR	5	0xfb5b20b0			
15 0 MISS	7	0xfb5b2010	D	ST	
16 0 WAIT	2	10	2		
17 0 MISS	8	0xfb5b20c0	D	LD	
18 0 WAIT	14	5	0		
19 0 WAIT_FOR	8	0xfb5b20c0			
20 0 MISS	9	0xfb5be0c0	D	ST	E
21 0 WAIT	132	72	0		
22 0 MISS	10	0x812cbfc0	I	LD	
23 0 WAIT	0	0	0		
24 0 MISS	0	0x812aafc0	D	LD	X
25 0 WAIT	0	0	0		
26 0 MISS	0	0x812aafc0	D	ST	X

Fig. 4.10 Trace example of one of the cores in a multi-core run of a simple vector addition program.

4.8 Concluding Remarks

The trace generator has been prototyped in gem5 and we are able to obtain traces with multi-core simulations and different SVE vector lengths. The tracing infrastructure has been validated with simple test cases and the traces are valid and suitable for replay. We have already generated and validated traces with up to 32 cores and with all possible vector lengths SVE supports. Some of these traces have been used to debug the RTL packet injector.

During the development of the infrastructure, some bugs have been found affecting the code of gem5. Also, some corner cases have been encountered while tracing the benchmarks. For example, after a branch miss-prediction which already executed a load link, we needed to delete the events already traced. This increased the complexity of the infrastructure by adding an event list module that needed to manage all this situations and only print or write the events to the final trace-file if it was successfully committed.

One of the main limitations of the current infrastructure is the support for atomics operations. Unfortunately, this support is not expected to be added to gem5 in the near future and we have decided to only allow LL/SC for the moment, which can mimic its behavior. The plan would be to add the support of atomics to the trace if they are added to gem5 before the project ends in 2021. These operations are of special interest to the NoC due to the complexity it raises.

Finally, Table 4.15 shows the applications traced for the Mont-Blanc 2020 project. Only some applications have been traced in 32 cores due to timing and size constraints. In addition, *miniAMR* and *SWFFT* presented some issues with 16 threads and only one configuration was successfully traced.

Application	8 threads					16 threads					32 threads				
	scalar	128	256	512	1024	scalar	128	256	512	1024	scalar	128	256	512	1024
Grid	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓					
HACCKernels	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓					
HPCG	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓					
EOS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓					
HYDRO	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LTIMES	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓					
MiniAMR	✓	✓	✓	✓	✓			✓							
Stream-DOT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SWFFT	✓	✓	✓	✓	✓			✓							
VOL3D	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
XSBench	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓					

Table 4.15 Obtained traces for MB2020 applications.

Chapter 5

Integrating RTL Models inside Full-System Simulators

Existing Systems-on-Chip (SoCs) have become incredibly complex, incorporating a large number of hardware blocks in their designs. These blocks can range from application- or domain-specific accelerators to smaller designs that provide additional monitoring or security capabilities. Most of these hardware blocks are developed in isolation to be later integrated into multiple different SoCs with different requirements. Therefore, these modules are often tested and designed independently, without a comprehensive view of the target platform integrated in.

Current tools to perform functional testing and design space exploration analysis of these hardware blocks are typically simulation-based. The main drawback is that these simulation frameworks are restricted to *testbenches* that feed the interfaces of the block in isolation. As a consequence, they do not model all the potential interactions and restrictions that may arise when the hardware block is integrated into a complex SoC with a complete software stack.

Hence, there is a clear need for tools that enable testing the implemented functionality of these hardware blocks, but also in terms of the expected performance they will provide on an existing SoC design. Ideally, these tools should be able to deliver a comprehensive hardware/software ecosystem where all the main components of the SoC are present with a complete software stack. In addition, it would be advisable that RTL designs already implemented in HDLs such as Verilog or VHDL could be integrated into the simulated system with minimal effort, namely without requiring additional porting implementation efforts.

In this chapter, we introduce the gem5+RTL framework, a flexible infrastructure that enables easy integration of existing RTL models with the popular full-system gem5 simulator [17]. gem5+RTL enables to perform functional testing and design space exploration studies of existing RTL models on a full-system environment that models an entire SoC able to boot

Linux and run complex multi-threaded and multi-programmed workloads. We make the following contributions:

- We provide a framework that enables easy integration of existing RTL hardware blocks within a SoC for full-system simulations. The framework provides an *RTLObject* class with the necessary functionality to communicate with any of the SoC components via standard gem5 *timing ports* and *packets*. This class has a clean application interface with the actual RTL model, which is converted to a *C++* shared library by leveraging the Verilator toolflow [66].
- We demonstrate how to use the framework to integrate existing RTL models into a full-system simulated SoC. For this purpose, we employ an in-house Performance Monitoring Unit (PMU) design and the NVIDIA NVDLA accelerator. For the latter, we are able to integrate up to four NVDLA instances on the same SoC with eight cores and a multi-level cache hierarchy.
- We evaluate the main functionalities of the PMU and validate the results with the statistics obtained from the gem5 simulation, which enables early testing on a system with multiple hardware components connected to the PMU. In addition, we perform a design space exploration that integrates multiple NVDLA accelerator instances on an SoC that features different memory technologies. We find out that each NVDLA instance has to support at least 64 in-flight memory requests to perform well on the evaluated workloads; and that as the number of NVDLA instances increases, certain memory technology configurations become a bad design choice as they fail to deliver sufficient memory bandwidth.

This chapter is organized in the following way: Section 5.1 describes our framework to enable RTL models in gem5, while Section 5.2 presents two relevant use cases based on a PMU and the NVDLA accelerator. Section 5.3 explains the experimental methodology, and Section 5.4 presents our evaluation of the proposed use cases. Finally, Section 5.5 summarizes our conclusions.

5.1 gem5+RTL Framework

In this section, we introduce gem5+RTL, a flexible framework that enables simulation of RTL models inside a full-system software simulator like gem5. First, we provide a general overview of the framework (Section 5.1.1). Then, we describe in detail the three main components in gem5+RTL (Sections 5.1.2-5.1.4). Finally, we discuss some relevant features in the framework and provide some illustrative connectivity examples (Section 5.1.5).

5.1.1 General Overview

Figure 5.1 shows an overview of the gem5+RTL framework. This framework has three main components:

a. *RTL Model*: The first step consists in using Verilator to obtain a C++ model from an RTL model written in Verilog or SystemVerilog. The obtained C++ model is ready to be integrated in the SoC of gem5, and no big changes in the RTL model are expected.

b. *Shared Library*: After generating the C++ model of the RTL design, a wrapper to interact with it and gem5 is needed. To ease the integration, a template of the wrapper is provided. Then, the wrapper and the C++ model are combined into a shared library that is integrated in gem5.

c. *Gem5 simulator Framework*: Inside gem5, a generic framework is provided to ease the integration of a wide range of potential hardware designs, which may require different needs in terms of connectivity. To achieve this, a generic *RTLObject* class is defined. This object comes bundled with all the functionality needed to interact with a simulated gem5 SoC.

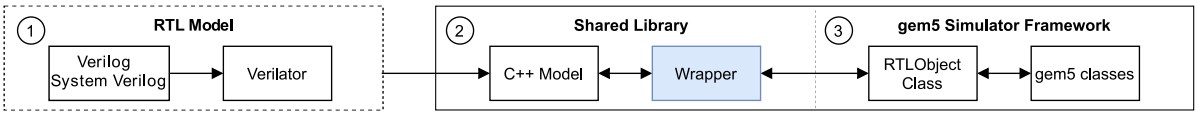


Fig. 5.1 The gem5+RTL framework has three main blocks: 1) RTL model; 2) a shared library that includes the C++ model generated with Verilator, and 3) gem5 extensions to communicate with the shared library.

In the following subsections, we describe such components in more detail.

5.1.2 RTL Model

The main objective of the gem5+RTL is to integrate RTL models in a full-system simulator without requiring to be modified. To do so, we make use of Verilator to convert an RTL model written in Verilog or SystemVerilog into a C++ model automatically. Verilator is relatively easy to use and works similarly as a compiler requiring to specify some arguments that can be checked on the documentation [65]. The next ones are the most important ones:

- **–top-module**: Specifies to Verilator the top module of the hardware design, which is only needed when there is more than one top-level module. It generates a warning otherwise. Also, by default, the generated top C++ module name is the same of the RTL module but starting with a letter *"V"* which can also be changed. For example, *Cache.sv* would become *VCache.cc*.
- **–compiler**: Specifies the compiler to be used. Currently, there is support for gcc, clang and msvc. In our experience, clang has been very useful for big hardware modules.
- **–sc | –cc**: Specifies whether the output module is in SystemC or C++.
- **{file.v}**: All the RTL files to be included.
- **-CFLAGS**: C++ compiler flags.
- **-O0|2|3**: Optimization level of the output C++ module. By default, no optimization is set because it can significantly add compilation time to the whole process. This can increase the performance of the output C++ module, which is very interesting once the integration is done.
- It can receive extra arguments to increase simulation performance via multi-threading (–threads), enable features like checkpointing (–savable) or enable the tracing of waveforms (–trace).

Like all the HDL simulators in the market, Verilator offers the possibility to output tracing waveforms from the C++ RTL model, both in FST and VCD format. This is extremely useful for debugging purposes. However, waveform traces can become prohibitively large. For this reason, we can enable and disable this feature in the output C++ module by an argument (–trace).

In addition, it also supports the FST format, which decreases the size of the final trace significantly at the expense of slower simulations (up to 10x of slowdown). We have found very slow the simulations with FST. However, it has been added the support to trace with a different

thread, which may increase the performance. If storage is a problem, we think it is better only to enable tracing for some time or to transform the waveform on VCD to FST format after the simulation is finished.[1].

Finally, one of the recent features added in version 4 of Verilator (end of 2018) was the support for multi-threaded parallelization of simulations. RTL simulations are very slow and can significantly increase the simulation time of the entire gem5 SoC even if using a simple RTL module. Thus, this feature can significantly speed-up the simulation time.

In our experience, multi-threaded has not been trivial to use, and no benefits have been gained after all the effort. It is a feature that depends on how much parallelization the design can offer and is improving in the new releases of Verilator. Making a better use of this feature remains as future work.

5.1.3 gem5+RTL Shared Library

After using Verilator to generate the C++ model of the RTL design, a wrapper to interact with the generated model and the gem5 SoC is needed. Similarly as a *testbench*, the wrapper interacts with the low-level interfaces of the RTL design and simplifies the integration with gem5.

The gem5+RTL framework requires to implement two additional simple functions in the wrapper: *tick* and *reset*. Both are employed by the generic gem5 infrastructure:

- **tick**: It is used to signal when a clock tick takes place. This function delivers the inputs to the RTL design and gets its outputs to gem5. All the input signals and output signals are bundled separately in structs to ease the small changes that new hardware modules will require.

- **reset**: It is used to reset the state of the modeled hardware as expected. The reset function does not take any argument as input neither output.

Apart from the functions mentioned before, the provided wrapper has additional functionalities like enabling and disabling in runtime the tracing of a waveform, or an AXI wrapper that handles memory communications through this type of protocol. In addition, a function to save the state of the RTL module could also be placed to enable doing checkpoints, quite useful to reproduce errors or to only trace some parts of an application. We think this feature is very interesting and remains as a future work.

The wrapper and the C++ RTL model are then combined into a shared library. The decision to use a shared library instead of a static one is due to significantly helping in the usability and

[1] This is a feature provided by gtkwave.

being able to compile independently gem5. It also draws a clear separation line between what needs to be provided for each RTL model and the generic framework that we provide on the gem5 side. As a result, it does not require recompilation if the shared library changes, when, for example, trying new features of Verilator or changing the RTL design.

5.1.4 Changes to Gem5 to Support RTL Models

Having created the shared library, on the gem5 side, we provide a generic framework to ease the integration of a wide range of potential hardware modules, which may require different needs in terms of connectivity. For this purpose, we have defined a generic *RTLObject* class ready to be used out-of-the-box. This object comes bundled with all the functionalities needed to interact with a simulated gem5 SoC, including predefined timing ports that enable seamless connectivity with the SoC. Depending on the complexity of the hardware module to integrate, minor adjustments will be needed. The following sections will explain in more detail some of these functionalities.

The RTLObject class is meant to act as an abstract layer between the actual RTL model being integrated and defined in the shared library and the gem5 simulated SoC. The following list describes the main features of this class:

- A set of gem5 timing ports to seamlessly connect with other gem5 objects. The class currently supports a total of four ports, two towards the CPU side and two towards the memory side of the SoC hierarchy. This includes all the necessary functionality to send and receive packets using gem5's timing infrastructure. Adding more ports is trivial, but most hardware blocks will find the provided ones sufficient.

- Functionality to connect to a TLB object for address translation. This can be an already existing object in the SoC or one specifically instantiated to be used by the integrated RTL model.

- A clock tick function that is called at the same frequency as the modeled SoC core objects. However, a parameter can be used to change the frequency with respect to the core objects in order to adapt to that of the modeled hardware block.

- Like all gem5 objects, it has an associated python class file used to instantiate and connect all the simulated objects within the SoC via the mentioned timing ports.

As mentioned before, the shared library wrapper must implement two functions, *tick* and *reset*. The function *tick* from the shared library is invoked from the RTLObject side inside its own tick event. The data necessary to perform a *tick* on the RTL model side is passed as a

parameter of the function via a void pointer to a predefined data structure. Similarly, the data produced on the RTL model side that is needed on the gem5 side is returned on another data structure at the end of the *tick* function. Therefore, the gem5 RTLObject needs to define these data structures and to have the necessary code to populate and consume their fields.

5.1.5 gem5+RTL Connectivity Examples

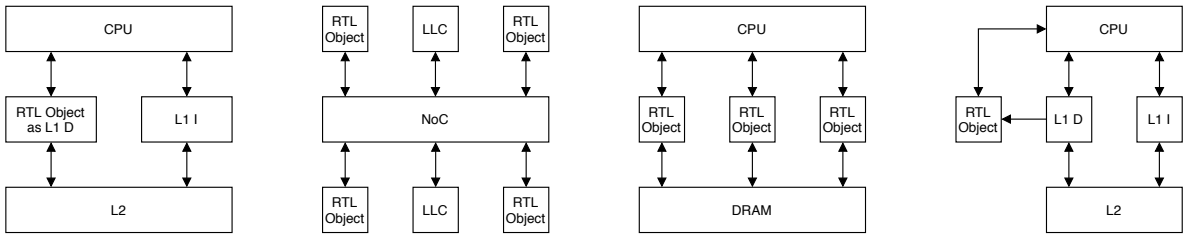


Fig. 5.2 Different connectivity options within the SoC of a simulated gem5+RTL system.

Figure 5.2 shows four examples illustrating different connectivity options within the SoC of a simulated gem5+RTL system. By using the provided infrastructure based on the gem5 standard ports, one could easily introduce the following RTL models in gem5:

(a) Use an RTL hardware block that models an L1 data cache and is connected to the core and L2 cache in gem5;

(b) Connect devices to the NoC such as accelerators, co-processors, or peripherals connected to off-chip components (e.g. JTAG, UART). Such devices could be coherent with the LLC;

(c) Hook near-memory or co-processor accelerators using an Input-Output Memory Management Unit (IOMMU);

(d) Connect additional blocks like a PMU that is connected to the CPU and the cache hierarchy to track different events.

The gem5+RTL has been designed to provide such flexibility in the inclusion of RTL models in the full-system simulator. In the following sections, we demonstrate the utility of the proposed framework by integrating two relevant use cases to a modeled SoC: a PMU model (see Figure 5.2 (d)) and the NVDLA (see Figure 5.2 (b)). The next section describes the implementation of such case studies in detail.

5.2 gem5+RTL Use Cases

Verification is an essential part of the process of developing a hardware design and probably the hardest and more time consuming. Different methods can be employed, but simulation-based techniques are the most popular. These methods are not as accurate as a real ASIC or an FPGA-emulated system, but it is a convenient first step to do functional verification of a design.

One of the problems when verifying a hardware block is that the validation environment is usually constrained to the block itself. Therefore, interactions that might only surface at the system level might not be stressed. For this reason, we use gem5 to model full-system setups and then insert RTL models to be able to check these interactions and test functionalities. In this context, our first use case is a PMU, for which we test its main functionalities and validate its results.

Our second use case focuses on using our framework to perform design-space exploration studies on the integration of hardware accelerators into the architecture of an existing SoC. Numerous hardware accelerators are designed in isolation and can be integrated in existing SoCs. However, their requirements in terms of, for example, memory bandwidth can determine how the integration occurs or the memory technology to be employed. To this end, we use our framework to perform a design space exploration analysis to connect multiple NVDLA accelerators into a SoC. Our tool enables to determine the right memory hierarchy configuration and to evaluate the trade-offs between different memory technologies.

5.2.1 Debugging RTL Models on a Full-System Environment

We have integrated an in-house PMU Verilog model with the simulated gem5 processor core. In the past, this synthesizable PMU has been integrated into the Cobham-Gaisler LEON3 [25] family of processors, which is a representative platform in the space domain. It provides a configurable number of counters to monitor events, programmable event thresholds to generate interrupts, and the possibility to have multiple cores connected to the PMU. It is interfaced through the Arm Advanced eXtensible Interface (AXI) protocol [11] for reading and writing the counters and modifying its configuration. The events are specified by a one-bit signal, rising up the signal on any cycle adds one to the event counter.

Figure 5.3 shows the necessary connections between the wrapper and the *C++* RTL model inside the shared library. The wrapper communicates with the *RTLObject* on the gem5 side via *structs* that are exchanged every tick. The *tick* function inside the wrapper receives an input struct with the necessary information to be sent to the PMU model, that is, the AXI read/write input and the event_enable[0-19] bits. Upon completion of the *tick* function, an output struct is

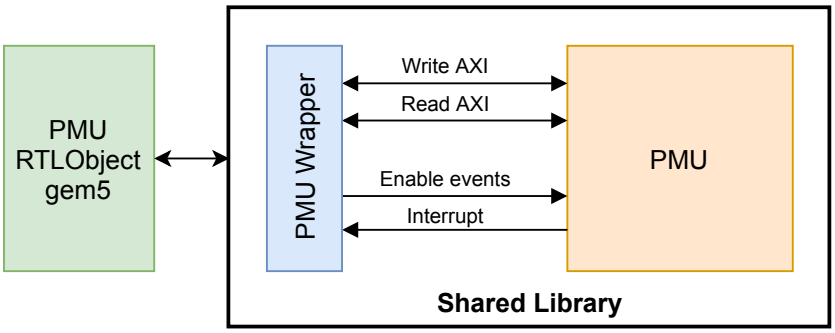


Fig. 5.3 PMU shared library connections.

filled so that the *RTLObject* receives the information returned on the AXI read/write channels and the interrupt signal.

To use the PMU, first, it needs to be configured by enabling the events to be monitored. Additionally, thresholds for specific events can be programmed by specifying a mask and a threshold value. This is done by writing to configuration registers via AXI commands. When a threshold is reached, the PMU generates an interrupt and resets the threshold counter. All of this configuration is usually wrapped up in a library for the final user to simplify the usability.

For evaluation purposes, we have connected a PMU to the commit instruction event of the core and to the L1D cache miss event. In the case of L1D misses, these can only happen once at any given cycle; however, the gem5 out-of-order core model we employ can commit up to four instructions per cycle. To properly configure the PMU, we have connected four event signals, making it possible to properly count the number of committed instructions at any given cycle.

Finally, we have also connected the cycle count as a PMU event. This allows us to generate periodic interrupts by setting a threshold for this event, which reports the different counter values. This use case enables us to test the PMU functionalities on a full hardware/software SoC environment encountering realistic problems such as the multiple commit instructions events.

5.2.2 Design-Space Exploration of the SoC Integration

NVDLA is an open-source accelerator created by NVIDIA that targets frequent inference operations such as convolutions, activations, pooling, and normalization. It is a flexible and scalable hardware design that is aimed at being the standard in the community. Even though it targets embedded systems, multiple NVDLA accelerators can coexist to target systems with different requirements. It was released open-source on 2018 with documentation, software source-code and an RTL design ready to be integrated on an ASIC or FPGA flow. Since 2018

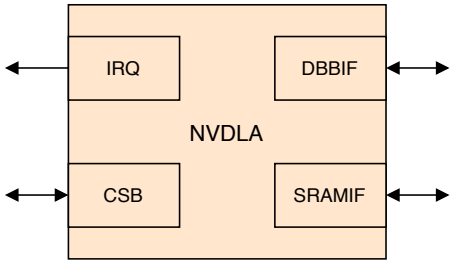


Fig. 5.4 NVDLA interfaces to connect the accelerator to a SoC.

some products from NVIDIA integrates instances of NVLDA in embedded platforms such as the jetson family of products [54, 55].

The implementation of the NVDLA accelerator provided is implem in Verilog. Also, different hardware configurations are available as a result of its flexibility, but only two are verified: *nv_large* and *nv_small*. The open-source release includes support for UVM verification and includes traces of real applications such as the GoogleNet and AlexNet Convolutional Neural Network (CNN) workloads.

Figure 5.4 shows the basic interfaces that NVDLA needs to connect to a SoC [53]:

- *Configuration Space Bus (CSB)* is a low bandwidth interface for the processor to configure the NVDLA accelerator.
- *External Interrupt (IRQ)* is a 1-bit signal that reports the completion of operations and errors.
- *Data BackBone (DBBIF)* is a high-bandwidth AXI4-compliant interface that is meant to be connected to a high-latency memory such as the main memory of a SoC.
- *SRAM Connection (SRAMIF)* is a secondary interface to connect with a small SRAM memory. The purpose of this small memory is to increase performance in systems that require high-performance.

To provide a clearer perspective of the interfaces, Figure 5.5 shows how the NVDLA hardware block is integrated into the gem5+RTL framework. The NVDLA release comes bundled with Verilator support and includes wrapper classes developed by NVIDIA. These classes provide AXI and CSB interfaces that, with minor modifications, can be used as part of the shared library wrapper that communicates with the *C++* RTL model obtained with Verilator as explained in Section 5.1.2.

Originally, the AXI wrapper models an ideal external main memory, which is now replaced by sending the requests to the *RTLObject* via the output struct. Then the *RTLObject* makes use of standard gem5 ports to send these requests to the main memory of the simulated gem5

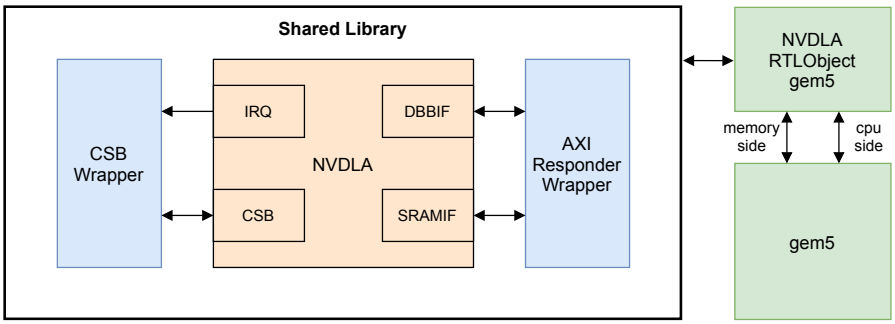


Fig. 5.5 Internal wrapper and gem5 connections for the NVDLA hardware block.

SoC. Similarly to the PMU, there is an input struct passed as a parameter to the wrapper *tick* function, with the data necessary to feed the interfaces to the NVDLA model, and an output struct with the data the RTLObject needs to communicate with the rest of the simulated SoC.

We have decided to connect both memory interfaces (DBBIF and SRAMIF) to the main memory subsystem of the simulated SoC. A better solution, and a possible extension of this work, could hook a proper SRAM such as a scratchpad memory to the *SRAMIF* interface.

Additionally, in a real system, an NVDLA accelerator should be connected to the IOMMU for programmability and security reasons. We have decided to bypass the IOMMU, as gem5 support for this hardware unit is still in early stages and there is little documentation. In addition, using an IOMMU requires a compliant device driver. The latter could be another possible extension to this work but will require much more time.

Finally, in late 2019, NVIDIA released the NVDLA compiler open-source [57]. Using this compiler would make perfect sense if the IOMMU is present, being able to mimic the correct flow through the software stack like a real platform such as the *Jetson-Xavier NX* [56] but on simulation.

5.3 Experimental Methodology

This section describes the configuration of the simulation infrastructure, the benchmarks employed, and the experiments performed to evaluate the two use cases with gem5+RTL.

5.3.1 gem5+RTL Configuration

The gem5+RTL framework has been configured to model an Arm-based SoC full-system environment that models the application, the operating system, and the architecture in detail. We simulate a contemporary SoC with eight out-of-order cores and a detailed multi-level memory hierarchy. The simulator is extended with the required RTLObject class to incorporate

different RTL models. Table 3.1 in Chapter 3 details the architectural parameters which are pretty standard only adding different memory models for design space exploration study.

5.3.2 Evaluated Benchmarks

PMU Case Study

We have created a simple benchmark composed of three sorting algorithms that exhibit different computational patterns: QuickSort [40], SelectionSort [35], and BubbleSort[12]. We execute each of these algorithms one after the other. Since QuickSort is significantly faster than the other two algorithms, we have used an array size of 30,000 elements for this algorithm and 3,000 elements for SelectionSort and BubbleSort. Between each of them, we insert a 1 millisecond sleep call to easily identify the end of each application phase when reading the performance counters. This will be seen as clear a drop of commit events or L1D misses.

It is worth mentioning that to not benefit any algorithm due to reusing the data on the D-Cache, each algorithm has a different copy of the array to sort. Adding to this, the given array is randomly created and different ones have been generated. Then, for each of the arrays generated, an execution has been made, although we will only show one execution in Section 5.4 of one array generated.

In the PMU, we set a threshold to generate interrupts every 10,000 cycles. When these interrupts occur, we record the events monitored by the PMU in order to extract Instructions-Per-Cycle (IPC) and Misses-Per-Kilo-Instruction (MPKI) metrics, which we then compare with the ones gem5 generates from its statistics over the same 10,000 cycle intervals.

NVDLA Case Study

We program the accelerators by running a simple user-level C++ application on the simulated SoC host cores. This application loads an NVDLA application trace into main memory, containing NVDLA instructions and data, and then signals the accelerator to start. Then, the application waits until the accelerator finishes execution.

When the NVDLA starts, the NVDLA RTLObject reads the trace loaded in main memory, and with the help of an auxiliary class called TraceLoader, it sends all the commands to the accelerator. This class has the needed functionality to read the provided workload traces and translates them into CSB operations that the CSB Wrapper will handle. We evaluate two application traces provided by NVDLA: a small memory-intensive convolution, *sanity3*; and the second convolution of the GoogleNet CNN pipeline, which has more computations and uses 3x3 filters, *GoogleNet*. These are representative of the kind of workloads that are expected to run on NVDLA hardware.

5.3.3 Performed Experiments

PMU Case Study

We have evaluated the functional behavior of the PMU with the sorting benchmark mentioned above. We gather the PMU counter values every 10,000 cycles and plot IPC and MPKI through time compared to gem5 statistics. Since reading requires more than one cycle, possible misses of events can occur.

We have also evaluated the simulation time overhead when adding the evaluated PMU RTL model to gem5. The reported execution times are the average over three simulations. We run our sorting benchmark on the standalone gem5 simulated SoC (*gem5*), using gem5 and the PMU RTL model (*gem5+PMU*), and *gem5+PMU* with waveform tracing enabled (gem5+PMU+wf).

NVDLA Case Study

We have performed the design space exploration study with a focus on the memory hierarchy. To do this, we have run an experimental campaign that takes into account the following parameters:

- *Maximum in-flight requests*: The maximum number of permitted in-flight memory requests from an NVDLA accelerator to the memory hierarchy.
- *Main memory technologies*: We have chosen DDR4, GDDR5, and HBM, being some of the most relevant nowadays.
- *Memory channels*: Number of memory channels for DDR4. We use a fixed quad-channel configuration for GDDR5, and HBM has a fixed number of eight channels per stack.
- *Number of NVDLA instances*: We have evaluated SoC systems that integrate 1, 2 and 4 NVDLA accelerator instances, reaching higher levels of stress to the memory system.

The goal is to determine the right amount of memory bandwidth necessary to feed a particular number of NVDLA accelerators and to see the trade-offs each of the memory technologies offer.

We have also evaluated the simulation time overheads in a similar manner than in the PMU. However, in this case we measure the execution time when running Verilator simulations with the provided NVDLA wrapper (*verilator*), gem5 with NVDLA using a perfect memory model (*gem5+NVDLA+perfect*), and gem5 with NVDLA and the DDR4 memory (*gem5+NVDLA+DDR4*).

5.4 Evaluation

In this section, it is described the evaluation performed using the gem5+RTL framework. First, we evaluate the PMU functionalities, and then perform a design-space exploration study using the NVDLA accelerator RTL model.

5.4.1 PMU Functional Evaluation

As described in Section 5.3, we evaluate the PMU functionality by registering a number of hardware events: (i) committed instructions, (ii) L1D cache misses, and (iii) cycles. To gather the values of all the mentioned counter events, we set a threshold for the cycles counter to generate an interrupt every 10,000 cycles, in a similar manner as a regular timer of a system. Also, every 10,000 cycles, we take a snapshot of the statistics gem5 keeps to be able to compare the results obtained with the PMU with respect to those in gem5.

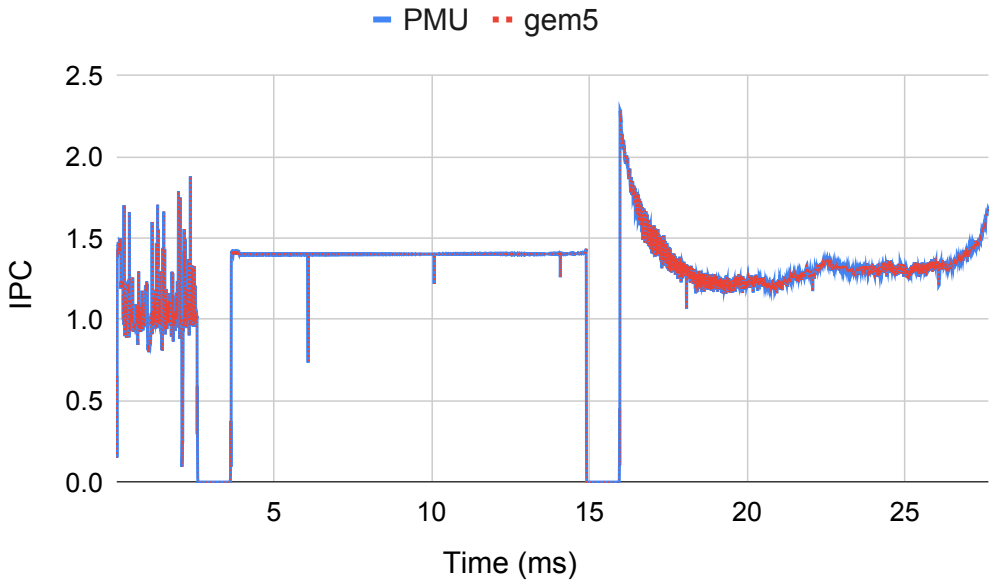


Fig. 5.6 IPC measurements over time (ms) for the PMU and gem5 statistics on three sorting kernels separated by 1ms sleep.

Figure 5.6 shows IPC (y-axis) across time (x-axis) for a benchmark that executes three sorting kernels. The solid line represents the measurements done by the PMU, while the dotted line represents those recorded by gem5 statistics. As can be seen in the figure, both measurements are reporting the same IPC values. However, looking at the data, we observe minor differences because of two reasons: (i) due to the 1-cycle delay of the PMU to record the

events, and (ii) because after an interrupt, a reset takes place, which means some events are lost during the few cycles the reset takes.

Although these discrepancies mentioned before are negligible and no visible differences are observed in the figure, gem5+RTL enabled us to study these interactions and determine the exact number of events lost due to the reset process. This is useful information for the PMU designers, who may take some action if the reset process hinders the target resolution of the PMU unit for certain threshold values.

Furthermore, it could also show bottlenecks on the communication with the PMU. For example, if the process of reading the events counter was too large, the gem5+RTL would enable research on a better solution in communication. This insight is only visible when simulating the whole software stack in a full Soc.

Looking at the actual IPC curves, we can easily distinguish the three sorting algorithms in the plot, as there is a 1*ms* sleep call between them, visible with an IPC of 0. As expected, QuickSort is clearly faster compared to SelectionSort and BubbleSort, which take a similar amount of time in this case.

Next, Figure 5.7 shows MPKI (y-axis) across time (x-axis). Similarly to the previous chart, we show measurements performed by the PMU and gem5 every 10,000 cycles. Again the measurements almost match with negligible differences due to the same reasons mentioned above. In particular, we observe that QuickSort stresses the memory hierarchy more than the other two sorting algorithms as it displays higher MPKI values.

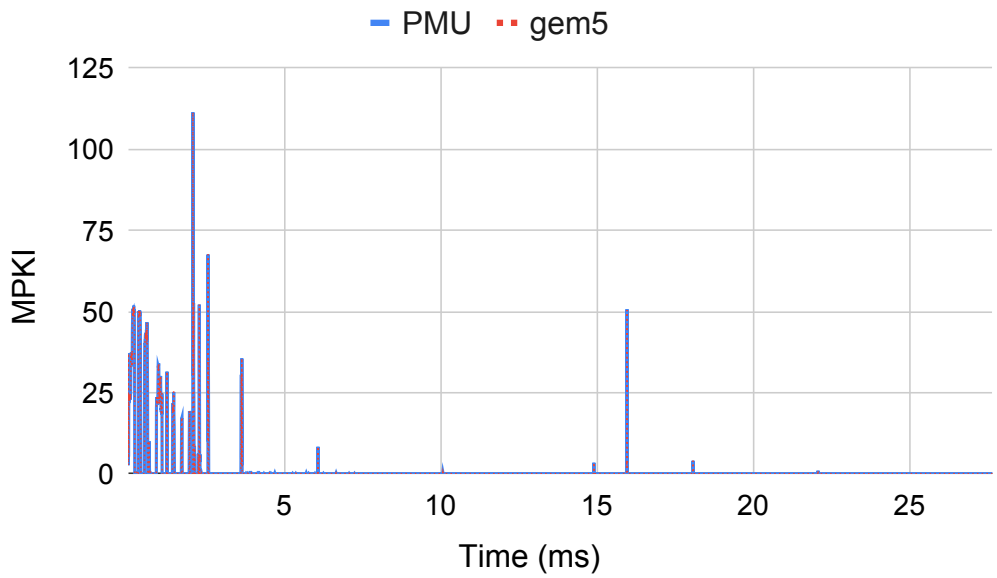


Fig. 5.7 MPKI measurements over time (ms) for the PMU and gem5 statistics on three sorting kernels separated by 1ms sleep.

Time overhead study

We have performed a time overhead study with the PMU that can be resumed on Figure 5.8; it shows the simulation time overhead of gem5+RTL normalized to a gem5 execution without the attached PMU RTL model over three different array sizes[2] (3, 30 and 60 thousand elements). We show results for a simulation that includes the PMU RTL model (*gem5+PMU*), and one that in addition has the waveform tracing support enabled (*gem5+PMU+wf*).

The overheads when adding the PMU model are of up to 1.24× without waveform support. gem5+RTL is able to integrate the generated cycle-accurate *C++* RTL model and simulate it with manageable overhead. This result is expected as the PMU RTL model is a relatively small hardware block with limited interaction with the whole SoC design. However, the overhead significantly increases to up to 7.27× when waveforms are enabled. Therefore, the use of waveforms needs to be minimized to avoid hefty simulation times.

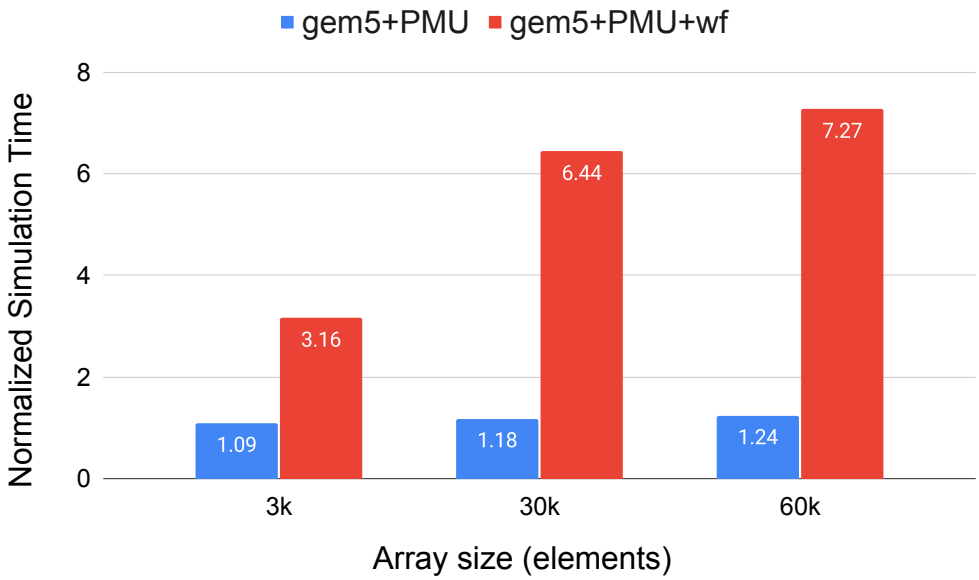


Fig. 5.8 Simulation time overhead when using gem5 and the PMU RTL model (*gem5+PMU*) and with waveform tracing enabled (gem5+PMU+wf) normalized to a gem5 execution without PMU. Simulations with three different array sizes (3, 30 and 60 thousand elements).

Finally, we have tried to improve the performance of the PMU C++ Model adjusting the flags in Verilator. First, multi-threaded option has been tried, although Verilator has been unsuccessful in obtaining a parallel design, giving an error of insatiability. This is probably due to being a very small design that cannot benefit from this relatively new feature. Second, different versions of Optimization flag (1,2,3,s) have been tried, being unexpectedly very useful,

[2]This size corresponds to the QuickSort and the rest of algorithms have the size divided by 10.

and the best option has been used for the final analysis explained before. Lastly, different types of waveform have been also tried, giving the best performance in execution time the vcd format.

5.4.2 NVDLA Design Space Exploration

Having seen the evaluation of the PMU, next, we evaluate the different trade-offs when integrating the NVDLA accelerator into an exiting SoC. To do this, we use two different workloads: *Sanity3* and *GoogleNet*. As described in Section 5.3, the parameters of the study include:

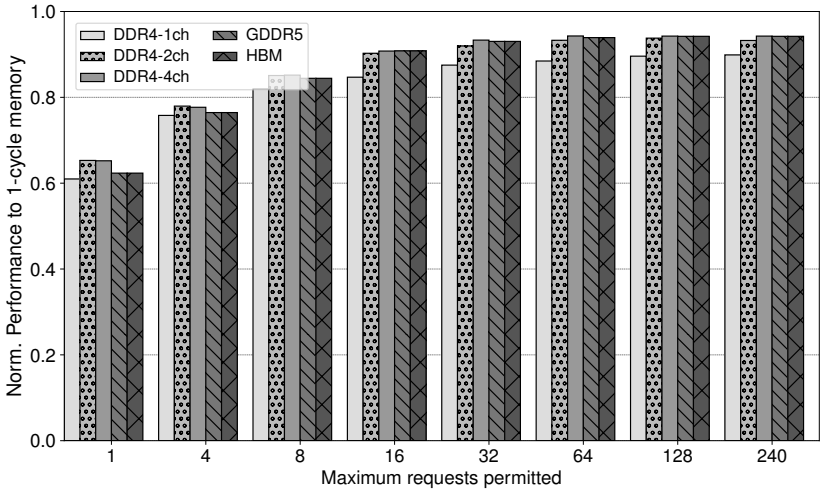
- The number of allowed in-flight memory requests for each NVDLA accelerator
- The number of NVDLA instances integrated into the SoC
- The main memory technology employed in the SoC

The number of allowed in-flight requests ranges from 1 to 240, the number of NVDLA instances from 1 to 4, and the different main memory configurations are: DDR4 with 1 channel, DDR4 with 2 channels, DDR4 with 4 channels, a four-channel GDDR5, and an HBM stack. These memory configurations offer different memory bandwidth capacities, as described in Table 3.2 on Chapter 3.

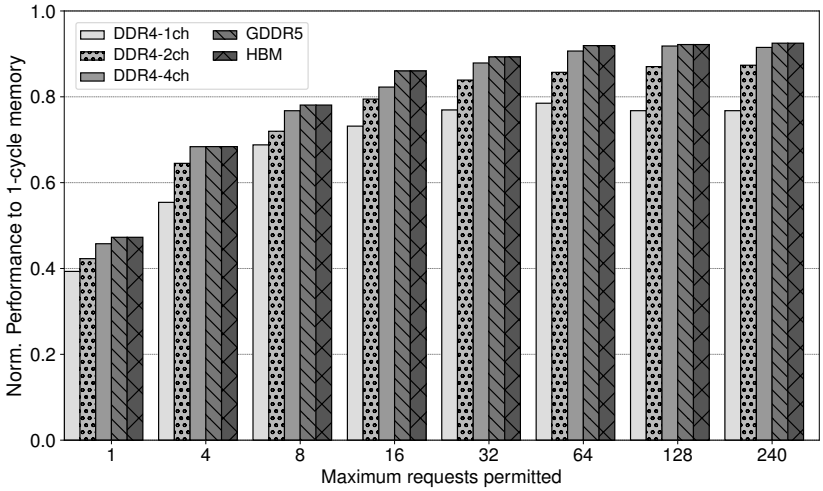
Figures 5.9 and 5.10 on the next pages show the performance results normalized to an ideal 1-cycle main memory with 1 (a), 2 (b), and 4 (c) NVDLA accelerators for *GoogleNet* and *Sanity3*, respectively. On the (y-axis) we have the performance normalized to an ideal 1 cycle-latency memory, and on the (x-axis) we have the different maximum requests permitted from the NVDLA to the memory. The figures are a bar chart with the bars grouped per memory request, and each memory configuration has a different bar-pattern. Finally, values close to one represent scenarios where memory contention is not affecting the performance of the NVDLAs.

As can be seen in *GoogleNet*, when employing one NVDLA accelerator (Figure 5.9 (a)), all memory technologies perform similarly, providing sufficient bandwidth to feed the accelerator. The only exception is DDR4-1ch, which falls a bit behind with 16 or more maximum in-flight requests.

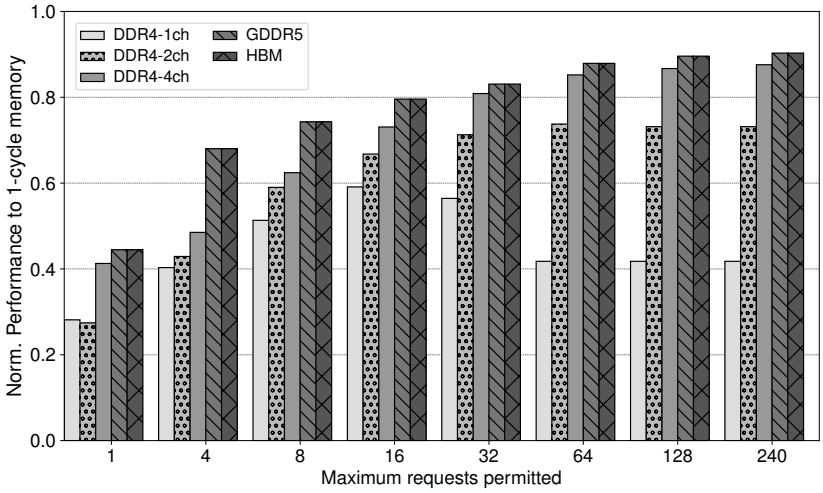
In *Sanity3*, this observation holds as the performance drops significantly with DDR4-1ch (see Figure 5.10 (a)). Even the DDR4-2ch and DDR4-4ch configurations fail to deliver comparable performance with respect to the GDDR5 and HBM configurations for 16 and 32 maximum in-flight requests. Nonetheless, if a maximum of 240 in-flight requests are allowed, the DDR4-2ch configuration offers competitive performance, and a system with such a memory configuration should be enough to host a single NVDLA accelerator.



(a) Performance of the system with a single NVDLA accelerator.

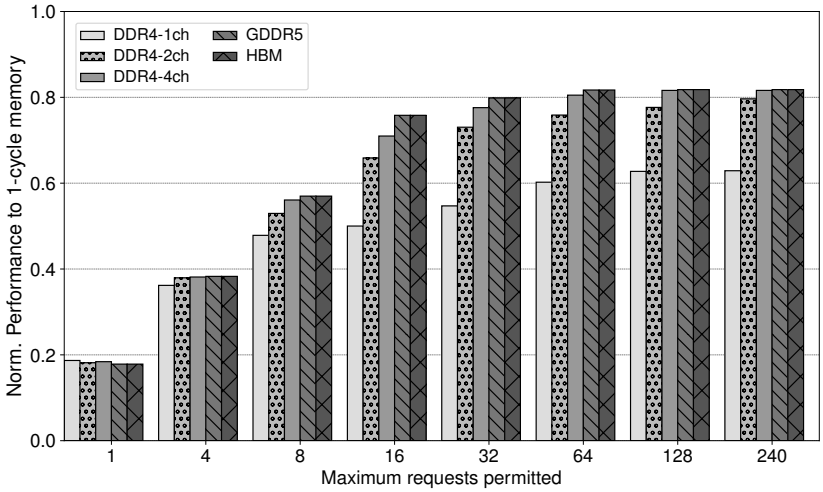


(b) Performance of the system with two NVDLA accelerators.

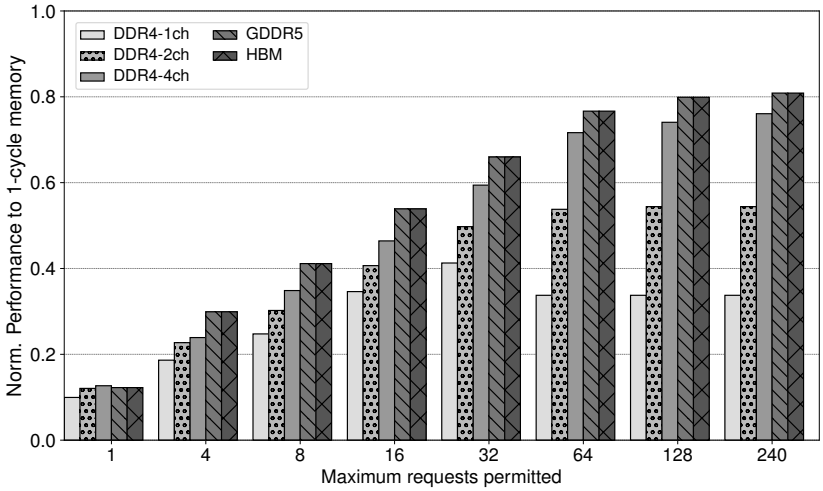


(c) Performance of the system with four NVDLA accelerators.

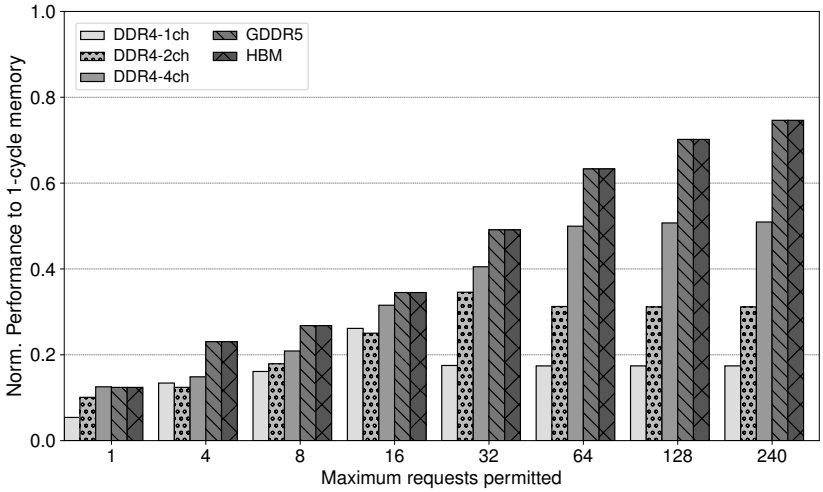
Fig. 5.9 Design-space exploration using the GoogleNet benchmark with 1, 2, and 4 NVDLAs. Normalized to an ideal 1-cycle main memory system.



(a) Performance of the system with a single NVDLA accelerator.



(b) Performance of the system with two NVDLA accelerators.



(c) Performance of the system with four NVDLA accelerators.

Fig. 5.10 Design-space exploration using the Sanity3 benchmark with 1, 2, and 4 NVDLAs. Normalized to an ideal 1-cycle main memory system.

When integrating two NVDLA accelerators (Figures 5.9 and 5.10 (b)), allowing at least 64 in-flight memory requests is a mandatory requirement to avoid significant performance degradation. The *GoogleNet* benchmark requires at least DDR4-4ch to attain the same performance as the high-bandwidth memory configurations. In the case of *Sanity3*, even with DDR4-4ch there is a noticeable performance degradation with respect to GDDR5 and HBM.

For the latter scenario, SoC designers need to decide whether the performance increase provided by GDDR5 and HBM memory technologies is worth the cost with respect to DDR4-4ch. gem5+RTL helps SoC designers make informed design decisions by evaluating an existing hardware RTL model on a full-system environment that would be very hard to replicate in existing simulation-based testing environments. To put things into perspective, the Jetson AGX Xavier Developer Kit includes 2 NVDLA accelerators and has 137GB/s of peak memory bandwidth [55]. The DDR4-4ch, GRRD5, and HBM configurations have 74.96GB/s, 112GB/s, and 128GB/s, respectively.

Finally, with four NVDLA accelerators (Figures 5.9 and 5.10 (c)), the number of maximum requests needs to be 240 to avoid performance degradation on the *Sanity3* benchmark. In this scenario, the use of high-bandwidth memory is recommended as DDR4-ch fails to deliver competitive performance. Even the GDDR5 and HBM technologies see a performance drop with respect to the 2 NVDLA accelerators from 0.81 to 0.74. Therefore, architecting an SoC with four NVDLA accelerators requires higher bandwidth technologies such as HBM2 in order to fully utilize the available hardware.

We observe performance degradation when increasing the number of memory requests that the NVDLA can send with the DDR4-1ch memory configuration. This behavior can be seen in Figures 5.9 (c), 5.10 (b), and 5.10 (c). We have confirmed that this is due to severe memory contention on the memory controller side. This in another example of the insight we can gain when employing the gem5+RTL framework.

Finally, we have also made a study with two additional benchmarks that can be seen in Appendix B.

Time overhead study

We have also performed a time overhead study, in a similar manner to the one of the PMU. Figure 5.11 shows the given time overhead of gem5+RTL normalized to a standalone Verilator simulation employing the wrapper that NVIDIA provides [3], namely *nvdla.cpp*[3]. This is compared to two gem5 simulated systems that integrate a single NVDLA accelerator, one simulating an ideal memory like the one used as a baseline in the design-space exploration eval-

[3]We have modified this C++ class to have less debug printing and to disable waveform tracing, significantly improving simulation time.

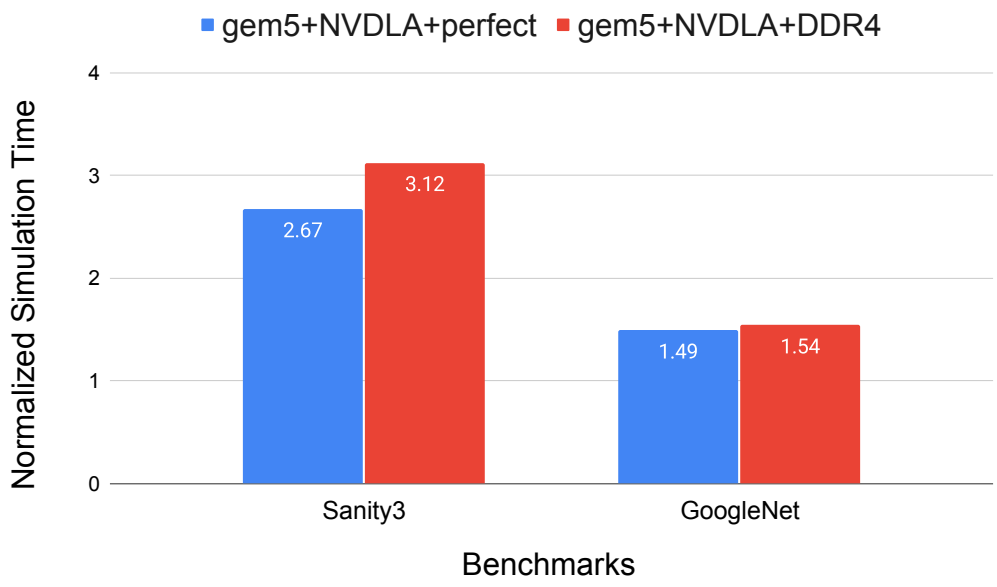


Fig. 5.11 Simulation time overhead of gem5+RTL normalized to a standalone Verilator simulation with a single NVDLA accelerator. *gem5+NVDLA+perfect* has an ideal memory, while *gem5+NVDLA+DDR4* uses the DDR4-4ch configuration.

uation (*gem5+NVDLA+perfect*), and another one using the DDR4-4ch memory configuration (*gem5+NVDLA+DDR4*).

Compared to the standalone Verilator execution, gem5+RTL takes up to 3.12× more time. The *Sanity3* benchmark has a higher overhead because the actual benchmark runtime is shorter, which makes the portion of time devoted to loading the trace into memory more prevalent. The standalone Verilator execution does not have this step as it reads the trace directly. Therefore, the simulation time overhead with larger benchmarks like *GoogleNet* is actually reduced to just 1.54×.

These latter simulation time overheads are expected as a full-system SoC with a DDR4 main memory model is now simulated. We also observe that simulation time does not increase significantly when going from the ideal memory system to the DDR4 configuration. This is because the amount of memory level parallelism is abundant in this benchmark, therefore memory access latencies can be hidden. Similarly, we have not seen big differences when changing the memory technology from DDR4 to GDDR5 or HBM.

We have also tried scenarios with tracing of a waveform enabled. A typical size of a given waveform was in the range of ten Giga Bytes (GB) for the sanity3 and hundred of GB's for GoogleNet. Having a limitation of space, we didn't continue studying tracing these enormous waveforms, although we could expect a significant increment of time overhead.

In addition, we have also tried to improve the C++ NVDLA model performance in a similar way than in the PMU. In this case, the multi-threaded option was available but non-significant improvements were obtained. Also, tracing with a separate thread was working with some improvement, but using the different optimizations levels made the best impact in the final performance. However, in this case, *Os* optimization level was the best one due to trying not to increase the size of the final binary, which was already quite big ($> 300MBs$).

To conclude, we strongly believe that these overheads are manageable and allow gem5+RTL to perform design-space exploration studies that can provide valuable insight when integrating hardware blocks into an SoC. Moreover, we strongly believe that the performance of the C++ Model can be improved and remains as an interesting small extension to this work.

5.5 Concluding Remarks

In this chapter, we have introduced a flexible framework that enables simulation of RTL models inside a full-system software simulator, gem5. The proposed framework enables to incorporate Verilog and SystemVerilog models anywhere on the simulated SoC design, which is able to boot Linux and run complex workloads.

As a result, we can evaluate two relevant use cases: a PMU and the NVDLA accelerator into a multi-core SoC. Thanks to the presented framework, we can test the RTL model features and perform an early-stage design space exploration of the entire SoC design.

We have used gem5+RTL to test the PMU functionalities and study potential interactions that arise from interfacing with an SoC. We have been able to observe and quantify small event count discrepancies that the reset functionality of the PMU introduces with respect to the measurements performed in gem5.

In addition, we have evaluated the integration of up to four NVDLA accelerator instances, finding that certain memory technologies might not deliver sufficient bandwidth to feed the accelerators. gem5+RTL helps SoC designers make informed design decisions by evaluating RTL models on a full-system environment that would be very hard to replicate in existing simulation-based testing environments like the one employed by Verilator.

To conclude, this framework enables early-design exploration of RTL Models inside a full SoC running in a regular software stack. This is particularly useful in early stages of a design and to check the integration when developing in isolation some RTL models. It cannot compete with FPGA solutions in performance, but its flexibility and being ready to use it without much effort can be a good trade-off to use it in parallel. Furthermore, we think Verilator is introducing new features at a very good pace, and its performance could increase even more.

Chapter 6

Conclusions and Future Work

On the first part of this thesis, the work has consisted on designing a trace specification, and later creating an infrastructure inside the gem5 full-system simulator to generate applications traces. The idea is to make traces of real applications in simulation per core to later be able to mimic its behavior with a traffic injector that only replays these traces. The final goal is to emulate the largest NoC possible by abstracting the cores with simple trace-based traffic injectors to reduce the size occupied in the emulation platform and put more *cores*.

The trace infrastructure has ended up being more challenging than expected and multiple problems have been found and solved. In particular, when the tests with the RTL packet injector begun, multiple issues raised. Some of them were trivial to solve like the repetition of WAIT_FOR events but other like managing the LL/SC were more difficult to solve.

This development has been a very interesting task as it has increased my knowledge of gem5, Out-of-Order architecture and debugging corner cases. One of the most beautiful corner cases found was related to deleting speculative memory operations already in-flight when executing an incorrect branch prediction. In addition, I have been in touch with different teams during the development to discuss bugs, define specifications and explore new extensions.

The trace generator has been prototyped in gem5 and we are able to obtain traces with multi-core simulations and different SVE vector lengths. The tracing infrastructure has been validated with simple test cases and the traces are valid and suitable for replay. We have already generated and validated traces with up to 32 cores and with all possible vector lengths SVE supports. Some of these traces have been used to debug the SVE RTL packet injector developed in the context of the Mont-Blanc 2020 European project.

In the current infrastructure, one of the main limitations is the support for atomics operations. Unfortunately this support is not expected to be added to gem5 in the near future and we have decided to only allow LL/SC for the moment, which can mimic its behavior.

The current situation of the Mont-Blanc 2020 project at October 2020 is the beginning of testing the emulator platform with simple traces obtained from this work. The NoC is still under development and the traces are still not working on it. However, this task is expected to be finished by 2021. Afterwards, the plan is to integrate the NoC in the European processor being developed at the European Processor Initiative (EPI) project.

On the second part of this thesis, the work has focused on the integration of RTL models inside full-system simulators. The resulting framework offers early design exploration inside a full SoC with seamless effort on the integration of RTL models. We have evaluated two relevant use cases: a PMU and the NVDLA accelerator into a multi-core SoC. Thanks to the presented framework, we can test the RTL model features and perform an early-stage design space exploration of the entire SoC design.

We have used gem5+RTL to test the PMU functionalities and study potential interactions that arise from interfacing with an SoC. We have been able to observe and quantify small event count discrepancies that the reset functionality of the PMU introduces with respect to the measurements performed in gem5.

In addition, we have evaluated the integration of up to four NVDLA accelerator instances, finding that certain memory technologies might not deliver sufficient bandwidth to feed the accelerators. gem5+RTL helps SoC designers make informed design decisions by evaluating RTL models on a full-system environment that would be very hard to replicate in existing simulation-based testing environments like the one employed by Verilator.

It has been a challenging work due to the usage of large HDL codes with tedious compilation procedures that lead to certain compilation issues. For instance, there was a big stall due to a problem with the G++ linker, as it was not able to link large object files (>300MB) in a reasonable amount of time. This ended up being a blocking factor, and the change to clang was crucial.

6.1 Future Work

With respect to the first contribution of the Master Thesis, we identified the following interesting future work:

- Add the support of atomics to the trace and to the trace infrastructure.
- Research on the generation of synthetic workloads. This novel technique is still an open research problem. Making synthetic traces that match real applications can be challenging and is very interesting to reduce, for example, the size of the traces, which can be prohibitively large. For instance, some traces in this work have ended up being as

big as 16 GB in binary format for a given application. Hence, we could apply some of the ideas that SynFull [15] proposes. For example, make applications models with our trace format that can be placed on RTL and hence reduce the hassle of sending the huge traces to a FPGA.

- Validate the traces and the tool-flow in the final demonstrator using the Veloce platform with a NoC of 16 and 32 cores.

With respect to the second contribution of the Master Thesis, we identified the following interesting future work:

- Improving the connectivity of the NVDLA with gem5, making use of a IOMMU, and then executing the operations with the real kernel provided by NVIDIA. This will also enable a better study with more benchmarks.
- Adding more RTL models and explore, for example, interesting re-programmable hardware that can be placed on the pipeline.
- Add more features to the framework, for example, allow checkpointing of RTL models connected to the regular checkpoints of gem5.
- Make a better study of which optimizations can be applied to Verilator to improve the final performance of the generated C++ model.
- Add more memory models like scratchpads to offer more flexibility.
- Add support for VHDL, the other well-known RTL language used in industry. There is a current open-source tool called GHDL [34] that offers a similar behavior to Verilator.

Probably, one of the most interesting extensions would be the usage of the IOTLB to enable coherent accelerators. There is a clear concern of security in today's designs and the way we connected the accelerator lacks the latest methods used in industry. The standard way to connect any accelerator that is outside the CPU, like in the Xavier SoC, is using the IOMMU unit. This piece of hardware is in charge of solving memory petitions from outside the CPU but managed by the OS, which can protect coherent parts of memory. Currently, gem5 offers limited support for these kinds of units and we have decided to not follow this path due to its complexity.

6.2 Contributions and Publications

The work performed in this Master Thesis has contributed to various publications, code repositories and project deliverables:

a. Publications:

[DCIS2020] J. Abella, C. Bulla, G. Cabo, F. J. Cazorla, A. Cristal, M. Doblas, R. Figueras, A. González, C. Hernández, C. Hernández, V. Jiménez, L. Kosmidis, V. Kostalabros, R. Langarita, N. Leyva, G. López-Paradís, J. Marimon, R. Martínez, J. Mendoza, F. Moll, M. Moretó, J. Pavón, C. Ramírez, M. A. Ramírez, C. Rojas, A. Rubio, A. Ruiz, N. Sonmez, V. Soria, L. Terés. O. Unsal, M. Valero, I. Vargas, L. Villa, ***"An Academic RISC-V Silicon Implementation Based on Open-Source Components."*** Conference on Design of Circuits and Integrated Systems (DCIS), 2020. **Paper accepted**.

[DATE2021] G. López-Paradís, A. Armejach, M. Moretó, ***"gem5+RTL: A Framework to Enable RTL Models Inside a Full-System Simulator"***. "Design, Automation, and Test in Europe" (DATE), 2021. **Under submission**.

b. Code contribution to lagarto, lagarto-lowrisc and gem5 git repository.

c. Contribution to European Projects: Mont-Blanc 2020 and EPI in the writing of some deliverables. Contribution to the DRAC project in the development of an in-order core.

Finally, the plan is to make the gem5+RTL framework open-source once the submitted publications is accepted in a conference.

References

[1] Agile analog approach. https://www.agileanalog.com/about/approach, Last Accessed: Oct 2020.

[2] Cadence verification website. https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html, Last Accessed: Oct 2020.

[3] Nvdla github webpage. https://github.com/nvdla/hw, Last Accessed: Oct 2020.

[4] Questasim. https://www.mentor.com/products/fv/questa/, Last Accessed: Oct 2020.

[5] Sifive introduces industry's first open-source chip platforms. https://www.sifive.com/press/sifive-introduces-industrys-first-open-source-chip-platforms, Last Accessed: Oct 2020.

[6] Synopsis vcs. https://www.synopsys.com/verification/simulation/vcs.html, Last Accessed: Oct 2020.

[7] ALCORN, P. Intel's 7nm is broken, company announces delay until 2022, 2023. https://www.tomshardware.com/news/intel-announces-delay-to-7nm-processors-now-one-year-behind-expectations, Last Accessed: Oct 2020.

[8] ANANDTECH. Hot chips 31: Tesla solution for full self driving car (2019). https://www.anandtech.com/show/14766/hot-chips-31-live-blogs-tesla-solution-for-full-self-driving, Last Accessed: Oct 2020.

[9] ARDESTANI, E. K., AND RENAU, J. Esesc: A fast multicore simulator using time-based sampling. In *High Performance Computer Architecture, 2013 IEEE 19th International Symposium on* (2013), pp. 448–459.

[10] ARGOLLO, E., FALCÓN, A., FARABOSCHI, P., MONCHIERO, M., AND ORTEGA, D. Cotson: infrastructure for full system simulation. *ACM SIGOPS Operating Systems Review 43*, 1 (2009), 52–61.

[11] ARM. Amba axi and ace protocol specification. https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf, Last Accessed: Oct 2020.

[12] ASTRACHAN, O. Bubble sort: An archaeological algorithmic analysis. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (New York, NY, USA, 2003), SIGCSE '03, Association for Computing Machinery, p. 1–5.

[13] AZIZYAN, G., MAGARIAN, M. K., AND KAJKO-MATSSON, M. Survey of agile tool usage and needs. In *2011 Agile Conference* (2011), pp. 29–38.

[14] BACHRACH, J., VO, H., RICHARDS, B., LEE, Y., WATERMAN, A., AVIŽIENIS, R., WAWRZYNEK, J., AND ASANOVIĆ, K. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference* (New York, NY, USA, 2012), DAC '12, Association for Computing Machinery, p. 1216–1225.

[15] BADR, M., AND JERGER, N. E. Synfull: Synthetic traffic models capturing cache coherent behaviour. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)* (2014), pp. 109–120.

[16] BECK, K., BEEDLE, M., VAN BENNEKUM, A., COCKBURN, A., CUNNINGHAM, W., FOWLER, M., GRENNING, J., HIGHSMITH, J., HUNT, A., JEFFRIES, R., ET AL. The agile manifesto, 2001.

[17] BINKERT, N., BECKMANN, B., BLACK, G., REINHARDT, S. K., SAIDI, A., BASU, A., HESTNESS, J., HOWER, D. R., KRISHNA, T., SARDASHTI, S., SEN, R., SEWELL, K., SHOAIB, M., VAISH, N., HILL, M. D., AND WOOD, D. A. The gem5 simulator. *SIGARCH Comput. Archit. News 39*, 2 (Aug. 2011), 1–7.

[18] BLAZE. Blaze website. http://blaze.com, Last Accessed: Oct 2020.

[19] BUTKO, A., GARIBOTTI, R., OST, L., AND SASSATELLI, G. Accuracy evaluation of gem5 simulator system. In *7th International workshop on reconfigurable and communication-centric systems-on-chip (ReCoSoC)* (2012), IEEE, pp. 1–7.

[20] CARLSON, T. E., HEIRMAN, W., AND EECKHOUT, L. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *High Performance Computing, Networking, Storage and Analysis, 2011 International Conference for* (2011), pp. 1–12.

[21] CHEN, W., RAY, S., BHADRA, J., ABADIR, M., AND WANG, L. Challenges and trends in modern soc design verification. *IEEE Design Test 34*, 5 (2017), 7–22.

[22] CHEN, Y.-K., AND KUNG, S. Y. Trend and Challenge on System-on-a-Chip Designs. *Journal of Signal Processing Systems 53*, 1 (Nov. 2008), 217–229.

[23] CHIOU, D. The microsoft catapult project. In *2017 IEEE International Symposium on Workload Characterization (IISWC)* (2017), pp. 124–124.

[24] CHIOU, D., SUNWOO, D., KIM, J., PATIL, N. A., REINHART, W., JOHNSON, D. E., KEEFE, J., AND ANGEPAT, H. Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)* (2007), pp. 249–261.

[25] COBHAM-GAISLER. Leon3ft fault-tolerant processor. https://www.gaisler.com/index.php/products/processors/leon3ft, Last Accessed: Oct 2020.

[26] COLLETT, R. E. Executive session: How to address today's growing system complexity. In *DATE'10: Conference on Design, Automation and Test in Europe* (2010).

[27] CONG, J., FANG, Z., GILL, M., AND REINMAN, G. Parade: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2015), pp. 380–387.

[28] DECALUWE, J. Myhdl: a python-based hardware description language. *Linux journal*, 127 (2004), 84–87.

[29] DENNARD, R. H., GAENSSLEN, F. H., YU, H., RIDEOUT, V. L., BASSOUS, E., AND LEBLANC, A. R. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits 9*, 5 (1974), 256–268.

[30] FENG, L., LIANG, H., SINHA, S., AND ZHANG, W. Heterosim: A heterogeneous cpu-fpga simulator. *IEEE Computer Architecture Letters 16*, 1 (Jan 2017), 38–41.

[31] FOSTER, H. The weather report: 2018 study on ic/asic verification trends. https://semiengineering.com/the-weather-report-2018-study-on-ic-asic-verification-trends/, Last Accessed: Oct 2020.

[32] FOX, C. Intel's next-generation 7nm chips delayed until 2022. https://www.bbc.com/news/technology-53525710, Last Accessed: Oct 2020.

[33] GENBRUGGE, D., EYERMAN, S., AND EECKHOUT, L. Interval simulation: Raising the level of abstraction in architectural simulation. In *High Performance Computer Architecture, 2010 IEEE 16th International Symposium on* (2010), pp. 1–12.

[34] GINGOLD, T. Ghdl (2007). http://ghdl.free.fr/, Last Accessed: Oct 2020.

[35] GOETZ, M. A. Internal and tape sorting using the replacement-selection technique. *Commun. ACM 6*, 5 (May 1963), 201–206.

[36] GRAPHCORE. Graphcore website. http://graphcore.ai, Last Accessed: Oct 2020.

[37] GRASS, T., ALLANDE, C., ARMEJACH, A., RICO, A., AYGUADE, E., LABARTA, J., VALERO, M., CASAS, M., AND MORETO, M. MUSA: A Multi-level Simulation Approach for Next-Generation HPC Machines. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2016), pp. 526–537.

[38] HENNESSY, J. L., AND PATTERSON, D. A. A new golden age for computer architecture. *Commun. ACM 62*, 2 (Jan. 2019), 48–60.

[39] HIGHSMITH, J., AND COCKBURN, A. What is agile software development? *The Journal of Defense Software Engineering 15*, 10 (2002), 4–9.

[40] HOARE, C. A. R. Quicksort. *The Computer Journal 5*, 1 (01 1962), 10–16.

[41] HOGAN, J. Hogan compares palladium, veloce, eve zebu, aldec, bluespec, dini. http://www.deepchip.com/items/0522-04.html, Last Accessed: Oct 2020.

[42] HOLTHAUS, M. J. Intel supply letters customer. https://newsroom.intel.com/wp-content/uploads/sites/11/2019/11/intel-supply-letter-customers.pdf, Last Accessed: Oct 2020.

[43] JOUPPI, N. P., YOUNG, C., PATIL, N., PATTERSON, D., AGRAWAL, G., BAJWA, R., BATES, S., BHATIA, S., BODEN, N., BORCHERS, A., BOYLE, R., CANTIN, P.-L., CHAO, C., CLARK, C., CORIELL, J., DALEY, M., DAU, M., DEAN, J., GELB, B., GHAEMMAGHAMI, T. V., GOTTIPATI, R., GULLAND, W., HAGMANN, R., HO, C. R., HOGBERG, D., HU, J., HUNDT, R., HURT, D., IBARZ, J., JAFFEY, A., JAWORSKI, A., KAPLAN, A., KHAITAN, H., KILLEBREW, D., KOCH, A., KUMAR, N., LACY, S., LAUDON, J., LAW, J., LE, D., LEARY, C., LIU, Z., LUCKE, K., LUNDIN, A., MACKEAN, G., MAGGIORE, A., MAHONY, M., MILLER, K., NAGARAJAN, R., NARAYANASWAMI, R., NI, R., NIX, K., NORRIE, T., OMERNICK, M., PENUKONDA, N., PHELPS, A., ROSS, J., ROSS, M., SALEK, A., SAMADIANI, E., SEVERN, C., SIZIKOV, G., SNELHAM, M., SOUTER, J., STEINBERG, D., SWING, A., TAN, M., THORSON, G., TIAN, B., TOMA, H., TUTTLE, E., VASUDEVAN, V., WALTER, R., WANG, W., WILCOX, E., AND YOON, D. H. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News 45*, 2 (June 2017), 1–12.

[44] KARANDIKAR, S., MAO, H., KIM, D., BIANCOLIN, D., AMID, A., LEE, D., PEMBERTON, N., AMARO, E., SCHMIDT, C., CHOPRA, A., HUANG, Q., KOVACS, K., NIKOLIC, B., KATZ, R., BACHRACH, J., AND ASANOVIC, K. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)* (June 2018), pp. 29–42.

[45] KHAN, A., VIJAYARAGHAVAN, M., BOYD-WICKIZER, S., AND ARVIND. Fast and cycle-accurate modeling of a multicore processor. In *2012 IEEE International Symposium on Performance Analysis of Systems Software* (2012), pp. 178–187.

[46] LEE, Y., WATERMAN, A., COOK, H., ZIMMER, B., KELLER, B., PUGGELLI, A., KWAK, J., JEVTIC, R., BAILEY, S., BLAGOJEVIC, M., CHIU, P., AVIZIENIS, R., RICHARDS, B., BACHRACH, J., PATTERSON, D., ALON, E., NIKOLIC, B., AND ASANOVIC, K. An agile approach to building risc-v microprocessors. *IEEE Micro 36*, 2 (2016), 8–20.

[47] LI, S., AHN, J. H., STRONG, R. D., BROCKMAN, J. B., TULLSEN, D. M., AND JOUPPI, N. P. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (2009), pp. 469–480.

[48] LIANG, T., FENG, L., SINHA, S., AND ZHANG, W. Paas: A system level simulator for heterogeneous computing architectures. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)* (Sep. 2017), pp. 1–8.

[49] LOCKHART, D., ZIBRAT, G., AND BATTEN, C. Pymtl: A unified framework for vertically integrated computer architecture research. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), pp. 280–292.

[50] MONEMI, A., TANG, J. W., PALESI, M., AND MARSONO, M. N. Pronoc. *Microprocess. Microsyst. 54*, C (Oct. 2017), 60–74.

[51] MOORE, G. E. Cramming more components onto integrated circuits. *Proceedings of the IEEE 86*, 1 (1998), 82–85.

[52] NIKHIL, R. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* (2004), pp. 69–70.

[53] NVIDIA. Integrator's manual (2018). http://nvdla.org/hw/v1/integration_guide.html, Last Accessed: Oct 2020.

[54] NVIDIA. Introducing jetson xavier nx, the world's smallest ai supercomputer. https://developer.nvidia.com/blog/jetson-xavier-nx-the-worlds-smallest-ai-supercomputer/, Last Accessed: Oct 2020.

[55] NVIDIA. Jetson agx xavier and the new era of autonomous machines. http://info.nvidia.com/rs/156-OFN-742/images/Jetson_AGX_Xavier_New_Era_Autonomous_Machines.pdf, Last Accessed: Oct 2020.

[56] NVIDIA. Jetson xavier nx. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/, Last Accessed: Oct 2020.

[57] NVIDIA. Nvidia deep learning accelerator. http://nvdla.org/primer.html, Last Accessed: Oct 2020.

[58] PELLAUER, M., ADLER, M., KINSY, M., PARASHAR, A., AND EMER, J. Hasim: Fpga-based high-detail multicore simulation using time-division multiplexing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture* (2011), pp. 406–417.

[59] SANCHEZ, D., AND KOZYRAKIS, C. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. In *ACM SIGARCH Computer Architecture News* (2013), vol. 41, pp. 475–486.

[60] SERVICES, A. W. Amazon EC2 F1 instances. https://aws.amazon.com/ec2/instance-types/f1/, Last Accessed: Oct 2020.

[61] SHACHAM, O., AZIZI, O., WACHS, M., QADEER, W., ASGAR, Z., KELLEY, K., STEVENSON, J. P., RICHARDSON, S., HOROWITZ, M., LEE, B., SOLOMATNIKOV, A., AND FIROOZSHAHIAN, A. Rethinking digital design: Why design must change. *IEEE Micro 30*, 6 (2010), 9–24.

[62] SHAO, Y. S., REAGEN, B., WEI, G.-Y., AND BROOKS, D. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (2014), ISCA '14, IEEE Press, p. 97–108.

[63] SHAO, Y. S., XI, S. L., SRINIVASAN, V., WEI, G., AND BROOKS, D. Co-designing accelerators and soc interfaces using gem5-aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Oct 2016), pp. 1–12.

[64] SHILOV, A. Intel delays mass production of 10 nm cpus to 2019. https://www.anandtech.com/show/12693/intel-delays-mass-production-of-10-nm-cpus-to-2019, Last Accessed: Oct 2020.

[65] SNYDER, W. Verilator documentation. https://www.veripool.org/projects/verilator/wiki/Manual-verilator, Last Accessed: Oct 2020.

[66] SNYDER, W. Verilator the fast free verilog simulator (2012). http://www.veripool.org, Last Accessed: Oct 2020.

[67] STEPHENS, N., BILES, S., BOETTCHER, M., EAPEN, J., EYOLE, M., GABRIELLI, G., HORSNELL, M., MAGKLIS, G., MARTINEZ, A., PREMILLIEU, N., REID, A., RICO, A., AND WALKER, P. The arm scalable vector extension. *IEEE Micro 37*, 2 (2017), 26–39.

[68] TAN, Z., WATERMAN, A., AVIZIENIS, R., LEE, Y., COOK, H., PATTERSON, D., AND ASANOVIC´, K. Ramp gold: An fpga-based architecture simulator for multiprocessors. In *Design Automation Conference* (2010), pp. 463–468.

[69] UBAL, R., JANG, B., MISTRY, P., SCHAA, D., AND KAELI, D. Multi2sim: A simulation framework for cpu-gpu computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2012), PACT '12, ACM, pp. 335–344.

[70] WAWRZYNEK, J., PATTERSON, D., OSKIN, M., LU, S.-L., KOZYRAKIS, C., HOE, J. C., CHIOU, D., AND ASANOVIC, K. Ramp: Research accelerator for multiple processors. *IEEE micro 27*, 2 (2007), 46–57.

[71] WILLIAMS, S. . Icarus verilog. http://iverilog.icarus.com/, Last Accessed: Oct 2020.

[72] ZHOU, G., ZHOU, J., AND LIN, H. Research on nvidia deep learning accelerator. In *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)* (2018), pp. 192–195.

Appendix A

Trace Statistics of the NoC Experiments

Tables A.1 and A.2 show a deeper analysis of all the trace statistics gathered in the preliminary study in a real NoC. Table A.1 has the interesting averages of the number of cycles waited for each benchmark and VL configuration. Table A.2 has the calculation of the *NoC* cycles adjusted to the maximum number of cycles that the packet injector waits (*Norm. NoC*).

APP NAME	VL	gem5 Sim C.	Norm. NoC C.	Ratio	AVG Wait C.	AVG Wait For C.
EOS	0	117,657,163	228,491,578	1.942011622	13.85455775	33.63824155
EOS	1	115,608,281	166,037,773	1.436210032	17.02516544	59.74928157
EOS	2	115,556,871	153,164,460	1.325446585	10.91999008	89.74772933
EOS	4	93,456,808	111,615,601	1.194301447	8.96406159	167.63127080
EOS	8	105,814,109	139,382,534	1.317239594	21.61722488	248.86424470
Grid+Wilson	1	200,370,130	205,030,041	1.023256515	31.71691842	37.48916510
Grid+Wilson	8	104,513,878	108,086,201	1.034180370	23.53216467	46.27796018
HACCKernels	2	169,300,391	170,120,735	1.004845494	20.43396905	50.03512623
HACCKernels	4	129,947,090	130,882,217	1.007196213	25.55307263	49.03024781
HACCKernels	8	53,083,603	53,751,572	1.012583340	22.53030303	51.83887119
HPCG	2	245,989,448	306,232,992	1.244902960	21.51202749	60.76418225
LTIMES	0	249,370,459	269,473,024	1.080613257	27.34615986	36.47103237
LTIMES	2	158,528,769	193,133,071	1.218284052	23.84177215	41.65876777
LTIMES	4	124,753,718	157,369,133	1.261438421	19.93278085	41.82896237
LTIMES	8	72,147,566	96,288,191	1.334600685	13.20224502	42.12450199

Table A.1 All columns after VL represent cycles and correspond to statistics obtained from the execution of the traces in gem5 and the NoC. Column ***gem5 Sim*** is the cycles needed to trace the given benchmark in gem5. Column ***Norm. NoC*** is the cycles needed to execute the trace in the NoC. Column ***Ratio*** is the division of the *normalized NoC* cycles by the *gem5 Simulated* cycles. Column ***AVG Wait*** is the average cycles stalled in *Wait* events. Column ***AVG Wait For*** is the average cycles waited between a memory petition and the *Wait FOR* event.

APP NAME	VL	NoC Sim	Wait_16	gem5 Sim	Wait	A=Wait - Wait_16	B= gem5 - Wait	Norm. NoC
EOS	0	**165,000,000**	54,165,585	**117,657,163**	109,678,673	55,513,088	7,978,490	**228,491,578**
EOS	1	**117,000,000**	66,570,508	**115,608,281**	115,504,138	48,933,630	104,143	**166,037,773**
EOS	2	**80,300,000**	42,692,411	**115,556,871**	115,521,478	72,829,067	35,393	**153,164,460**
EOS	4	**53,200,000**	35,041,207	**93,456,808**	93,433,615	58,392,408	23,193	**111,615,601**
EOS	8	**71,000,000**	37,431,575	**105,814,109**	100,453,506	63,021,931	5,360,603	**139,382,534**
Grid+Wilson	1	**28,700,000**	24,040,089	**200,370,130**	200,368,504	176,328,415	1,626	**205,030,041**
Grid+Wilson	8	**16,160,000**	12,587,677	**104,513,878**	104,509,940	91,922,263	3,938	**108,086,201**
HACCKernels	2	**1,640,000**	819,656	**169,300,391**	168,635,964	167,816,308	664,427	**170,120,735**
HACCKernels	4	**1,730,000**	794,873	**129,947,090**	129,875,129	129,080,256	71,961	**130,882,217**
HACCKernels	8	**1,540,000**	872,031	**53,083,603**	53,041,576	52,169,545	42,027	**53,751,572**
HPCG	2	**146,000,000**	85,756,456	**245,989,448**	245,896,744	160,140,288	92,704	**306,232,992**
LTIMES	0	**32,000,000**	11,897,435	**249,370,459**	249,367,520	237,470,085	2,939	**269,473,024**
LTIMES	2	**45,350,000**	10,745,698	**158,528,769**	158,525,845	147,780,147	2,924	**193,133,071**
LTIMES	4	**49,000,000**	16,384,585	**124,753,718**	124,750,389	108,365,804	3,329	**157,369,133**
LTIMES	8	**40,400,000**	16,259,375	**72,147,566**	72,145,613	55,886,238	1,953	**96,288,191**

Table A.2 All columns after VL represent cycles and correspond to statistics obtained from the execution of the traces in gem5 and the NoC. Column ***NoC Sim*** is the number of cycles needed to execute the trace on the NoC. Column ***Wait_16*** is the number of cycles that the packet injector waits in the NoC execution. Column ***gem5 Sim*** is the number of cycles needed to trace the given benchmark in gem5. Column ***Wait*** is the number of cycles that the trace specifies to wait. The calculation of **Normalized NoC Cycles** is the following: *Norm. NoC: NoC Sim + A + B*. This calculation tries to adjust the cycles obtained from the NoC execution by accounting for the avoided wait cycles in the NoC (A) and the overhead on the gem5 side due to wait_for and communication (B). This number can be comparable to *gem5 Sim*, but having avoided wait events affects the total execution time without being able to account for this deviation. Hence, an execution on the NoC side with the full wait events is needed to understand the trace execution in the NoC better.

Appendix B

Other Benchmarks Studied on the gem5+RTL Design Space Exploration

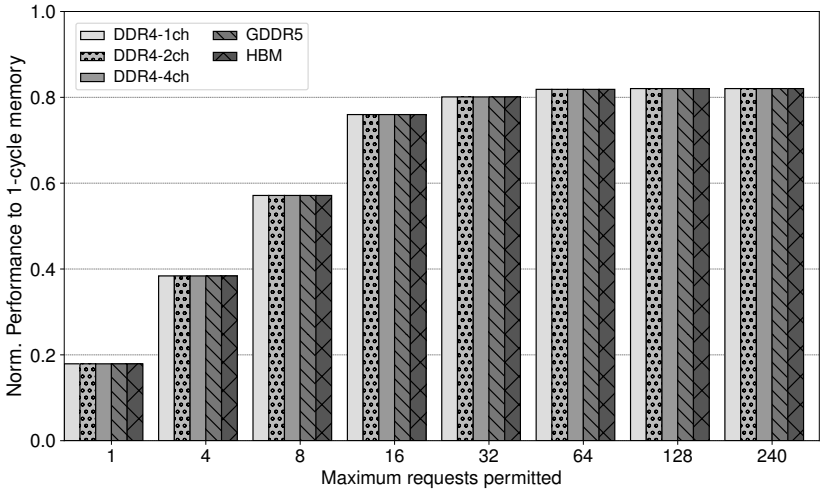
On the next two pages, we can se Figure B.1 and B.2 that are two extra benchmarks also studied in the evaluation of the gem5+RTL explained in Chapter 5, Section 5.4.

Figure B.1 is the study of a straightforward and short convolution benchmark. Due to the size of the benchmark and not being memory intensive, we can see no performance degradation when changing the memory configuration. The only difference that can be observed is the degradation of performance when the number of permitted maximum requests from the NVDLA to the memory is changed.

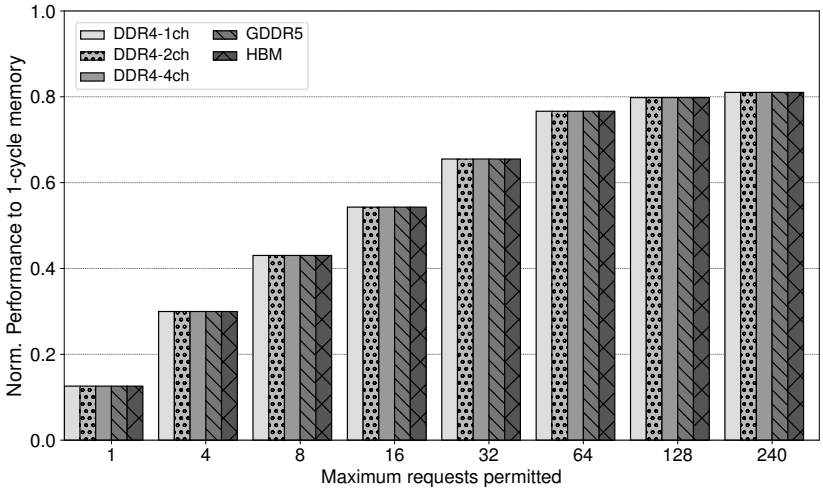
However, it is also interesting to study these very simple benchmarks because it could happen that, for example, in extensive benchmarks, there could be a phase with a very planar performance. Therefore, using these simple benchmarks could lead to co-design methods to improve the efficiency of the system in these phases as well.

Figure B.2 shows the study of a convolution of the *AlexNet* convolutional neural network (CNN). Unlike the last benchmark, here we can see performance degradation when changing the memory configuration. Also, we can observe the same insight of performance degradation from the evaluation in Chapter 5: DDR4-1ch fails to deliver enough bandwidth after 16 permitted in-flight requests on fig B.2b and fig B.2c.

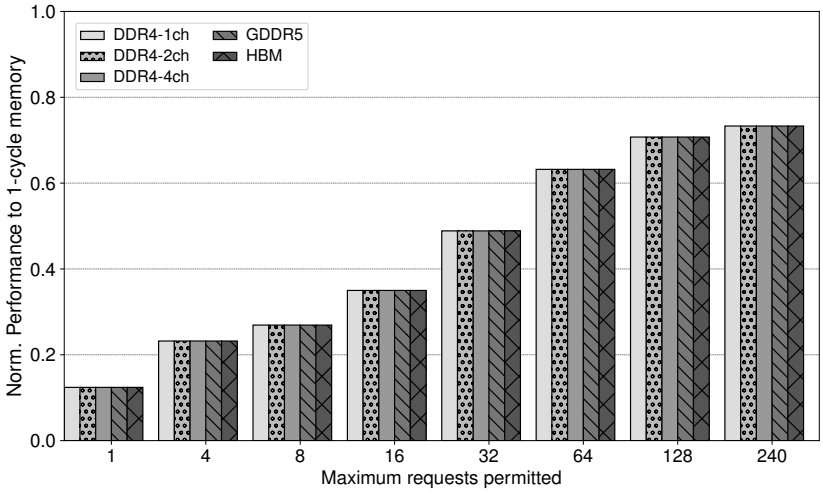
Adding to the latter, we can also observe the expected performance degradation when changing the maximum number of permitted memory requests. Finally, in fig B.2a we can see that DDR4-2ch and DDR4-4ch outperform GDDR5 and HBM in some situations. This is probably because in less than 32 maximum permitted memory requests, the benchmark is more subject to the latency than the bandwidth of the memory.



(a) Performance of the system with a single NVDLA accelerator.

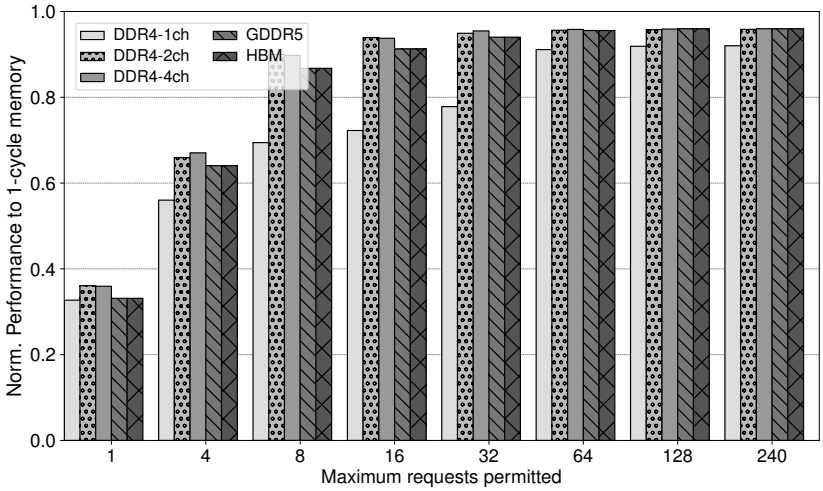


(b) Performance of the system with two NVDLA accelerators.

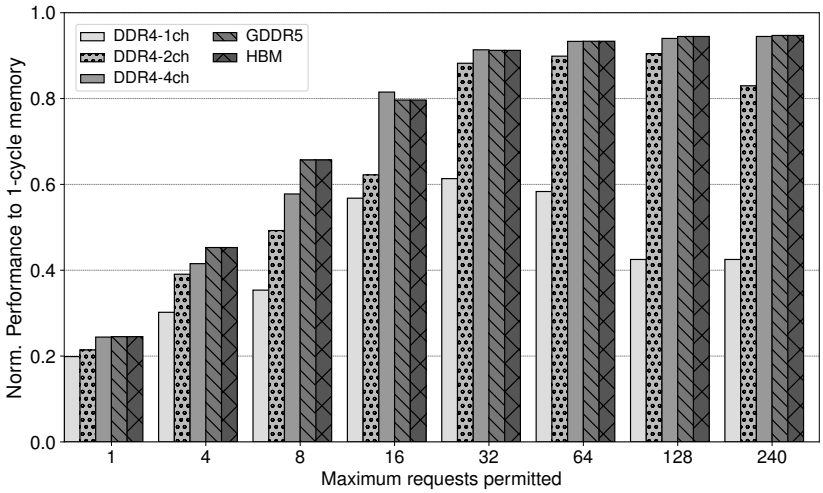


(c) Performance of the system with four NVDLA accelerators.

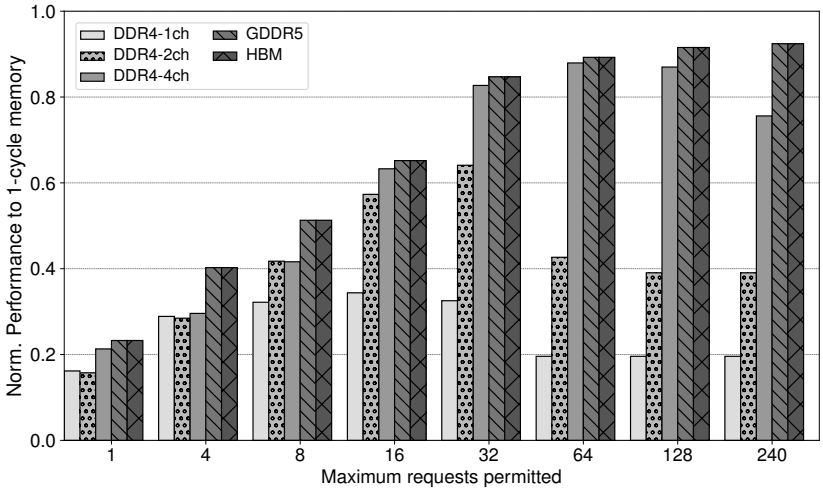
Fig. B.1 Design-space exploration using a simple convolution benchmark with 1, 2, and 4 NVDLAs. Normalized to an ideal 1-cycle main memory system.



(a) Performance of the system with a single NVDLA accelerator.



(b) Performance of the system with two NVDLA accelerators.



(c) Performance of the system with four NVDLA accelerators.

Fig. B.2 Design-space exploration using AlexNet benchmark with 1, 2, and 4 NVDLAs. Normalized to an ideal 1-cycle main memory system.