CHALMERS
UNIVERSITY OF TECHNOLOGY

Verification and Performance Evaluation Methodologies for Open-Source SoCs on FPGA Proof-of-Concept Platforms

Master's thesis in Embedded Electronic System Design

SIDDHARTHAN AZHAGUVEL

Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026

Master's thesis 2026

Verification and Performance Evaluation Methodologies for Open-Source SoCs on FPGA Proof-of-Concept Platforms

SIDDHARTHAN AZHAGUVEL

Department of Microtechnology and Nanoscience
Chalmers University of Technology
Gothenburg, Sweden 2026

Verification and Performance Evaluation Methodologies for Open-Source SoCs on FPGA Proof-of-Concept Platforms
SIDDHARTHAN AZHAGUVEL

Supervisor: Mehrzad Nejat, Department of Computer Science and Engineering
Company advisor: Ahsen Ejaz, InfiniNode Technologies AB
Examiner: Per Larsson-Edefors, Department of Microtechnology and Nanoscience

Master's Thesis 2026
Department of Microtechnology and Nanoscience
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Verification and Performance Evaluation Methodologies for Open-Source SoCs on FPGA Proof-of-Concept Platforms
SIDDHARTHAN AZHAGUVEL
Department of Microtechnology and Nanoscience
Chalmers University of Technology

Abstract

Verifying functional correctness and evaluating performance before committing a design to silicon is a resource-limited challenge, particularly for academic groups and small enterprises lacking access to large-scale simulation infrastructure or proprietary IP. This thesis proposes a methodology for functional verification and performance monitoring using open-source SoC platforms (as they are free) to build a proof-of-concept prototype to test it on an FPGA (as they are cheap and accessible). It combines a staged flow with a top-down performance modelling approach in which fine-grained hardware counter instrumentation is added only where an abstract model identifies a bottleneck. The methodology is evaluated through a case study on the open-source Pulpissimo platform, ported to a previously unsupported FPGA target (KCU105 board from AMD), across a single-core baseline and the same system augmented with a hardware MAC accelerator. For each system, an execution-time model adapted from the literature is mapped onto existing and newly-designed hardware counters, integrated into the platform's RTL, and verified in simulation before being validated on FPGA. The baseline model reproduces measured execution time within 11%, using instrumentation that costs under 2% of on-chip logic area. For the accelerator-augmented system, the model was extended to capture dispatch overhead that scales with offload granularity. Both models correctly attribute the dominant performance bottleneck: core-internal effects for the baseline, and SoC-integration effects once the accelerator is added. This instrumentation was possible because the platform is open-source: its RTL could be directly modified to add the required counters, which is not generally achievable with proprietary IP. Together, these results demonstrate a low-overhead, adaptable modelling methodology suited to the resource constraints faced by academic groups and small enterprises.

Keywords: Field Programmable Gate Array (FPGA), proof-of-concept (PoC), open source, system-on-chip (SoC), functional verification, performance monitoring

Acknowledgements

I would like to thank my academic supervisor, Mehrzad Nejat, for his valuable technical guidance throughout this project and for helping streamline the flow of this thesis. I am also grateful to my industrial supervisor, Ahsen Ejaz, for offering suggestions from a practical, industry-oriented standpoint that shaped this work. I would like to thank my examiner, Per Larsson-Edefors, for his feedback on the half-time report and for the opportunity to present this thesis. Finally, I would like to thank my friends and family for their emotional support throughout this thesis.

Siddharthan Azhaguvel, Gothenburg, September 2026

Contents

List of Figures

List of Tables

1

Introduction

Open-source hardware has transformed the landscape of system-on-chip (SoC) development in recent years. Open architectures such as RISC-V, alongside fully open-source SoC platforms like PULP [2], OpenPiton [3], and Chipyard [4], have democratized access to high-quality processor cores, interconnects, peripheral controllers, and verification collateral. These ecosystems enable universities, small companies, and research groups to design and prototype complex SoC architectures without relying exclusively on proprietary commercial intellectual property (IP). As a result, open-source SoC components serve as building blocks for rapid architectural exploration, experimental designs, and early prototyping—particularly when combined with field-programmable gate arrays (FPGAs), which remain the dominant platform for proof-of-concept (PoC) implementations. Notable commercial products leveraging RISC-V include SiFive's E6-A series and Codasip L730, illustrating the practical adoption of open-source designs in AI and other applications.

In the application-specific integrated circuit (ASIC) development cycle, FPGA-based PoC prototypes play a critical role between register-transfer level (RTL) development and tape-out. A typical flow involves: (1) specification and architectural exploration, (2) RTL development and simulation, (3) FPGA-based PoC prototyping for functional verification and software bring-up, and (4) full ASIC implementation and tape-out. FPGA prototypes allow designers to evaluate functional correctness, run software workloads, and perform preliminary performance analysis on real hardware long before committing to the costly tape-out phase. This stage is particularly essential for open-source SoCs, where design teams often lack access to proprietary verification frameworks and industrial-grade toolchains.

While simulation-based verification (e.g., Universal Verification Methodology (UVM)) is exhaustive and precise, it is limited by slow execution speeds, often several orders of magnitude slower than real-time. FPGA PoC prototyping addresses this by providing real-time execution, enabling validation of long-running workloads such as operating system booting, memory stress tests, or network benchmarks. However, this approach introduces challenges, including limited FPGA resources, long re-implementation times, and constrained signal visibility. Performance monitoring on FPGA prototypes also differs from ASIC implementations: while tools such as Integrated Logic Analysers (ILAs), synthesisable assertions, and custom counters provide monitoring capabilities, a performance counter is only as technology-independent as the parameter it measures. Where a parameter is guaranteed by an architectural protocol contract, such as a fixed-latency interconnect handshake, its

cycle count remains portable between FPGA and ASIC, though the target's achievable clock frequency may still differ. Where no such contract exists and a counter instead reflects how long the underlying physical implementation actually takes, its value can diverge between FPGA and ASIC targets [5].

Mature proprietary SoC ecosystems, such as ARM CoreSight [6], provide integrated debug infrastructures, embedded performance monitoring, and vendor-supported SDKs. Open-source SoC developers, however, must often build verification and monitoring support from scratch, carefully balancing available FPGA resources with functional and performance requirements.

This thesis aims to define a methodology that guides developers in leveraging open-source SoC ecosystems for FPGA-based prototyping, enabling both functional verification and performance monitoring in resource-constrained environments. It evaluates the methodology at the FPGA PoC stage alone. While it defines which resulting metrics would be expected to transfer to an eventual ASIC implementation, no ASIC-stage validation is performed as part of this work.

1.1 Related Work

Research efforts in open-source SoC verification have primarily focused on improving simulation-based flows, with limited emphasis on FPGA-based PoC verification. Verilator [7] introduced a fast, cycle-accurate simulation approach that has become widely adopted in the open-source community, but its applicability is constrained to simulation environments and offers no solution for observability or debugging on FPGA prototypes. Similarly, Schiavone *et al.* [8] proposed an open-source verification framework for RISC-V cores, demonstrating that simulation-driven infrastructures can be standardised and reused across designs. However, their methodology remains simulation-centric and does not address the resource, visibility, and iteration constraints inherent in FPGA-based verification of complex SoC components.

FPGA-assisted verification has been explored in several works. FERIVer [9] presents an FPGA-accelerated emulation framework for RISC-V RTL, showing that hardware execution can significantly speed up test campaigns. Nonetheless, the framework still depends on vendor-specific debug capabilities and does not propose a structured methodology for functional coverage or iteration management under FPGA resource limits. Szymkowiak [10] describes practical challenges in FPGA-based prototyping of a modern multiprocessor SoC (MPSoC), including limited debugging visibility and long synthesis cycles, reinforcing the need for systematic verification flows. Earlier work by Kim *et al.* [11] demonstrated FPGA-based verification of SoC-type CMOS image processors, but their approach is tightly coupled to the target application and lacks generality for broader SoC ecosystems. Jirsak *et al.* [12] proposed netlist-level instrumentation for coverage monitoring directly on FPGA, enabling partial visibility improvements, though with overheads and without addressing performance evaluation or design portability. Beyond SoC verification, Larsson-Edefors and Börjeson [13] demonstrated the use of FPGA-based emulation for real-time evaluation of complex system behaviour, highlighting the role of FPGA PoC platforms as an intermediate validation step when simulation alone is insufficient.

Performance monitoring for FPGA systems has been studied separately from verification. Schmidt *et al.* [14] introduced HwPMI, an extensible on-FPGA performance monitoring infrastructure, demonstrating the value of modular in-hardware instrumentation. Jafri [15] developed a centralised performance monitoring system that aggregates counters and metrics for system-level analysis, though the focus is on FPGA-based system design rather than SoC integration. While these works provide useful building blocks, they do not integrate performance monitoring with functional verification or address the need for resource-constrained methods suitable for open-source SoC PoC development.

Overall, prior research addresses functional verification and performance monitoring largely in isolation, and mostly outside the specific constraints of FPGA-based PoC prototyping for open-source SoCs. Simulation-centric verification frameworks [7, 8] do not address the reduced observability, limited resources, and slow iteration inherent to FPGA prototyping, while FPGA-assisted verification efforts [9, 10, 11, 12, 13] either depend on vendor-specific tooling, remain tied to a particular target application, or do not propose a structured, resource-aware methodology for functional coverage under FPGA resource limits. Separately, performance monitoring infrastructures for FPGA [14, 15] demonstrate the value of in-hardware instrumentation but are not developed with functional verification or open-source SoC integration in mind. To the best of our knowledge, no existing work provides a methodology that addresses both functional verification and performance monitoring specifically for FPGA PoC prototypes of open-source SoC components, under the scalability, portability, and resource constraints that such development typically faces.

1.2 Purpose and Goal

FPGA PoC prototypes introduce several constraints that complicate functional verification compared to RTL simulation. Any design change requires time-consuming synthesis and implementation, making iteration slow. FPGA resource limits restrict the amount of debug logic, synthesisable assertions, and ILA probes that can be added, and only a small subset of internal signals can be observed. This limited visibility, combined with the inability to freely instrument the design, makes debugging integrated open-source SoC components significantly more challenging than in simulation environments.

Performance evaluation on FPGA prototypes faces similar limitations. Resource constraints restrict the number of performance counters and monitors that can be instantiated, while re-implementation time limits the number of design and test iterations. Additionally, performance evaluation must be technology-independent and synthesisable to make them relevant to the later ASIC design and implementation.

Given these limitations, a central research question is raised:

How can functional verification and performance monitoring be performed efficiently on FPGA PoC prototypes built from open-source SoC components, while remaining scalable, portable, and feasible within limited development resources and time?

The **purpose** of this thesis is to explore this question and examine how verification and performance evaluation practices can be structured and adapted to meet the practical constraints of FPGA-based PoC development.

The overall **goal** of this thesis is to develop a scalable and portable methodology for validating FPGA PoC prototypes built using open-source SoC components. The methodology has two complementary objectives:

Functional Verification Objective: Define an efficient verification flow that supports functional testing of integrated open-source SoC components on an FPGA prototype.

As part of this objective, the thesis will survey verification metrics suitable for FPGA PoC validation. These include component-level and system-level functional coverage expressed in terms of exercised test scenarios rather than hardware description language (HDL) coverage, which is inherently simulation-centric. Additional metrics include the ability to run representative benchmarks or software workloads, and the robustness of the system under sustained operation. The survey will examine how these metrics can be practically measured on FPGA prototypes.

Performance Monitoring Objective: Establish a lightweight and technology-independent performance instrumentation strategy suitable for FPGA prototypes. This involves identifying performance metrics that are meaningful at the PoC stage, determining how they can be captured using synthesisable instrumentation, and understanding the trade-offs imposed by FPGA resource constraints.

FPGAs enable the use of powerful but temporary debug structures—such as ILAs—that consume significant logic and memory resources but can be removed or reconfigured across FPGA builds. Such mechanisms are generally unsuitable for ASICs due to the permanent silicon area and power overhead they would incur. This reconfigurability fundamentally differentiates how performance monitoring and debugging are approached in FPGA PoC environments versus ASIC designs. As part of this objective, the thesis surveys performance metrics aligned with PoC project goals, including cache behaviour (e.g., hit/miss events), interconnect data and coherence traffic (e.g., snoops), congestion indicators such as router buffer occupancy, and throughput-related measures. These metrics are classified based on whether they are (i) relevant and measurable on FPGA, (ii) relevant to ASICs but impractical to measure accurately on FPGA, or (iii) primarily FPGA-specific and intended only for PoC-level debugging. The methodology explores how FPGA reconfigurability can be exploited to selectively enable, disable, or rotate instrumentation across builds, balancing observability against resource usage while preserving portability to ASIC flows.

These methodologies are subsequently applied in a case study, in which an open-source SoC platform is selected and prototyped on an FPGA-based platform. The case study serves as a practical evaluation of the proposed approaches, enabling an assessment of their effectiveness and feasibility in a real-world development context.

1.3 Thesis Outline

Chapter 2 provides the necessary technical background, introducing the key concepts required for understanding the remainder of the thesis. Chapter 3 presents the literature concepts needed in support of the functional verification objective, and outlines the methodology proposed for the subsequent case study.

In a similar manner, Chapter 4 addresses the performance monitoring objective by discussing the relevant concepts, including performance modelling techniques, and detailing the proposed methodology to be applied in the case study.

Chapter 5 then presents the case study itself, which focuses on a single open-source SoC platform. The methodologies introduced in the preceding chapters are applied in order to evaluate their effectiveness and to assess the soundness of the proposed approaches.

Chapters 6 and 7 present the results and discussion, respectively. The outcomes of the case study are analysed, and conclusions are drawn regarding the effectiveness of the proposed methodologies, supported by quantifiable metrics.

Finally, Chapter 8 concludes the thesis and outlines directions for future work.

2

Technical Background

This chapter presents the necessary background knowledge required to understand the work of this thesis. It begins by introducing the architecture of a typical SoC, including its components and possible variations. Subsequently, the role of FPGA-based prototyping within the SoC design flow is discussed, along with the differences between implementing a design on an FPGA prototype and on an ASIC. The chapter then elaborates on the context of functional verification in FPGA prototyping, highlighting key verification objectives and associated challenges. Finally, background on performance monitoring is provided, including an introduction to performance models and the role of performance counters.

2.1 SoC Architecture Fundamentals

A modern system-on-chip, as the name implies, integrates an entire system onto a single piece of silicon. A typical SoC consists of processor cores, cache hierarchies, memory controllers, an interconnect for on-chip communication, and various peripherals. Figure 2.1 illustrates a representative SoC architecture.

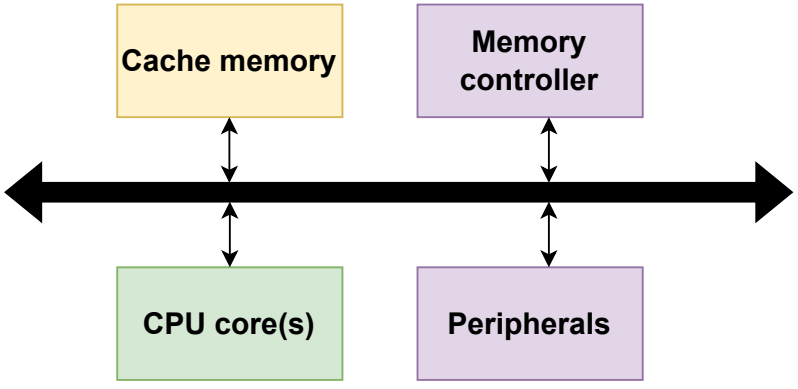

Figure 2.1: Typical SoC architecture

The processor, or central processing unit (CPU), typically refers to one or more cores, which may vary depending on application requirements. These variations include single-core or multicore designs, in-order or out-of-order execution, and single-issue

or multiple-issue architectures supporting multithreading. The cores generally interact with a cache hierarchy rather than directly accessing main memory (RAM). Cache accesses are significantly faster than off-chip memory accesses, thereby reducing core stall time during memory operations. Memory controllers, such as Double Data Rate (DDR) controllers, provide the interface to off-chip main memory. Main memory is typically located off-chip because the semiconductor manufacturing processes for memory differ from those used for processor chips.

Depending on system complexity, the interconnect may be implemented either as a shared bus or as a network-on-chip (NoC). Bus-based interconnects, such as those based on Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) interfaces, provide a shared communication medium where only one component can access the bus at a time, necessitating arbitration mechanisms. In contrast, a NoC consists of interconnected routers that forward packets between components. NoCs are typically employed in more complex systems comprising tens to hundreds of components, as they scale more effectively. They also allow multiple transactions to occur concurrently, with the network handling routing and delivery. Figure 2.2 compares bus-based and NoC-based communication architectures.

Peripherals enable communication with off-chip components, such as flash memory, and provide external interfaces for input/output operations, including Inter-Integrated Circuit (I2C), Universal Serial Bus (USB), and Universal Asynchronous Receiver-Transmitter (UART). These peripherals are generally based on standardised protocols and are therefore often implemented using pre-designed vendor IP blocks rather than being developed in-house. As a result, design effort is typically concentrated on the processor, cache hierarchy, and interconnect.

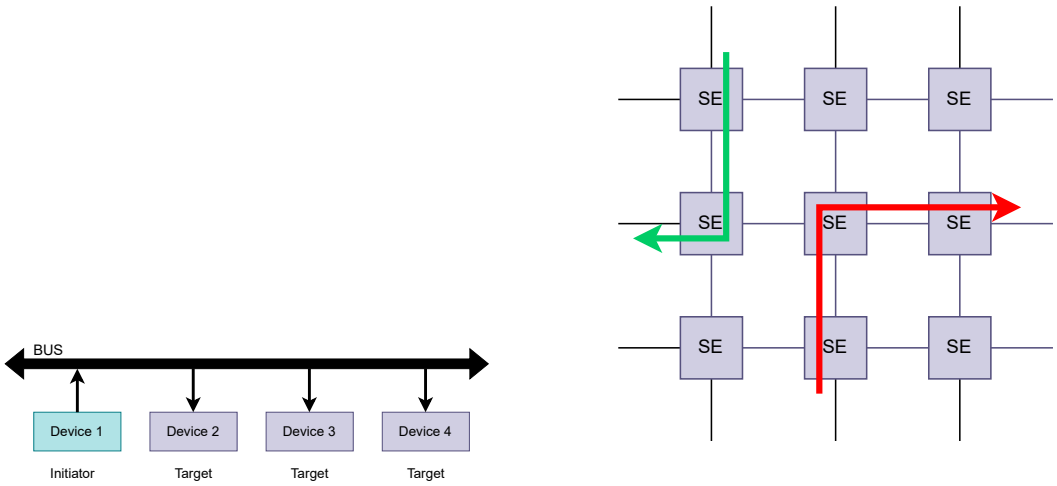

Figure 2.2: (a) Bus architecture with one master (initiator) and multiple slaves (targets) (b) A NoC architecture with switching elements (SE) or routers; the arrows show different data paths possible.

With the increasing demand for higher performance and the rise of artificial intel-

ligence (AI) applications, modern SoCs frequently incorporate custom accelerators. These accelerators provide specialised functionality with significantly higher performance and efficiency than general-purpose CPUs for targeted workloads. For example, AI accelerators are often designed to efficiently perform operations such as multiply-and-accumulate (MAC). In addition to accelerators, other processing units, such as graphics processing units (GPUs), may also be integrated on-chip to enable tighter coupling and improved performance. Systems that integrate multiple types of processing elements are referred to as heterogeneous architectures.

As discussed above, SoCs can vary widely in their architecture, and design teams may focus their research and development efforts on specific components to gain a competitive advantage. In this context, open-source SoC platforms provide a means to rapidly experiment with architectural variations and evaluate performance without the need to design an entire system from scratch. The following subsection provides an overview of some existing open-source platforms and their advantages.

2.2 Open-Source SoC Platforms

Traditionally, hardware IP has been proprietary and closely guarded. The introduction of open instruction set architectures (ISA) such as RISC-V has significantly accelerated research and development in open-source hardware. Designers can adopt base RISC-V processor implementations and customise them to meet specific requirements, rather than relying on proprietary IP from vendors such as Arm Ltd. or Intel. This has led to the emergence of open-source SoC platforms that enable full-system customisation.

Several notable platforms are briefly described below.

The PULP Platform (Parallel Ultra Low Power), developed by ETH Zürich and the University of Bologna, is an open-source platform for designing energy-efficient parallel computing systems. It is based on the RISC-V ISA and includes a range of hardware and software components to support parallel processing applications [16]. The platform integrates processor cores, memory subsystems, communication interfaces, and accelerators, along with supporting software and development tools.

OpenPiton, developed at Princeton University, is a tiled many-core framework designed for general-purpose, multithreaded workloads. It is based on the SPARC v9 ISA and employs a distributed directory-based cache coherence protocol across on-chip networks [3]. The platform provides configurability in both core and uncore components, and includes support for synthesis and back-end flows targeting both ASIC and FPGA implementations.

Chipyard, developed at the University of California, Berkeley, differs from fixed-IP platforms by providing a system-on-chip generator framework. In this approach, the RTL is generated based on user-defined configurations. Chipyard includes generators for key SoC components such as processors, cache hierarchies, interconnects, and accelerators. It is built around the Rocket Chip generator and, like PULP, is based on the RISC-V ISA [4].

These platforms provide reusable building blocks and flexible design frameworks;

however, they often lack standardised methodologies for FPGA-based prototyping, functional verification, and performance monitoring. This gap motivates the need for structured approaches, as explored in this thesis.

Open-source hardware IP is typically released under one of a small number of license families, which differ mainly in whether they impose obligations on derivative or integrated works. Permissive licenses (e.g. MIT, BSD, Apache 2.0) allow free use, modification, and redistribution, including in closed-source or commercial products, with minimal obligations beyond retaining the original copyright notice. Weak copyleft licenses (e.g. LGPL) require that modifications to the licensed IP itself remain open, but do not extend this requirement to a larger design that merely integrates the IP. Strong copyleft licenses (e.g. GPL, and the hardware-focused CERN-OHL-S) require that the complete design incorporating the licensed IP be released under the same license, which can be a significant constraint for commercial or closed integration. Table 2.1 summarises this distinction.

Table 2.1: Common open-source license categories and their implications for integration

License type	**Examples**	**Commercial/closed integration**
Permissive	MIT, BSD, Apache 2.0, Solderpad	Permitted, no reciprocal obligation
Weak copyleft	LGPL	Permitted; modifications to the IP itself must stay open
Strong copyleft	GPL, CERN-OHL-S	Requires the full integrating design to be open

2.3 FPGA-Based Prototyping

As part of any chip design process, having a prototype of the final product available to quickly validate design decisions is crucial for saving time and resources. FPGA-based prototyping provides a variety of use cases that benefit both hardware and software developers [17].

Firstly, it provides a platform for architectural exploration in the early stages of the design cycle. Because of its reconfigurability, different architectures can be implemented to verify simulation-model results against real hardware. Secondly, it provides a platform for software developers to verify that their software conforms to the programming model provided by the hardware team. Lastly, once software and hardware are nearing completion, the FPGA provides a platform for functional verification as "pre-silicon validation" — the first point at which the developed hardware and software stack interact on the same platform. Before this stage, hardware verification relies on simulation and emulation, which restricts the scope of what can be tested, while software relies on software models of the underlying hardware, which abstract away hardware intricacies such as timing.

In addition to the above, an FPGA prototype is useful for testing real-world effects, owing to its portability and its ability to function standalone without a host PC. For

example, a cellular modem SoC can be loaded onto an FPGA with its software held in flash memory and taken out of the lab environment to interact with real-world cellular signals. This provides high confidence in the design while also enabling analysis of real-world behaviour.

2.4 Performance Evaluation

With the increasing complexity of SoC architectures, it becomes essential to establish robust performance evaluation strategies to ensure that a system under design meets its intended performance targets in a systematic and meaningful manner. This process involves several key aspects, including the selection of an appropriate performance modelling approach, the choice of representative benchmarks, the identification of relevant performance metrics, and the design of infrastructure required to measure these metrics in hardware [18].

Given that an SoC comprises multiple interacting components, it is desirable to define a performance model that captures system behaviour in a mathematical form. This is typically achieved by decomposing the system into a set of parameters that can be quantified, either through simulation or direct measurement on hardware. The structure of the model depends on the specific performance metric of interest.

When performance is evaluated on hardware, specialised hardware counters, commonly referred to as performance counters, are used to record events within the system that correlate with the parameters of the model. The values obtained from these counters are then incorporated into the performance model to compute the desired metric. This process is typically repeated across multiple runs to obtain averaged results and improve measurement reliability.

The system under evaluation is usually exercised using targeted benchmarks. The choice of benchmark depends on the intended application domain of the system. For instance, the SPEC CPU benchmark suite is widely used for evaluating general-purpose processors, while benchmarks such as MLPerf and ULPMark target machine learning and embedded workloads, respectively. These benchmarks are standardised through extensive analysis of real-world workloads and are therefore considered reliable indicators of system performance in practical scenarios.

Example: Simple Completion Time Model

To illustrate the concept of performance modelling, consider a simplified system consisting of a single processing core, a basic interconnect, and a main memory [19]. A common performance metric for such a system is the total completion time required to execute a workload.

The total completion time can be expressed as the sum of three primary components: the computation time on the core, the latency incurred in accessing memory through the interconnect, and the memory access latency itself. This can be written as:

$$T_{\text{total}} = T_{\text{comp}} + T_{\text{bus}} + T_{\text{mem}} \tag{2.1}$$

where:

- T_{total} is the total execution (completion) time,
- T_{comp} is the computation time on the processor core,
- T_{bus} is the latency associated with transferring requests and responses over the interconnect,
- T_{mem} is the memory access latency.

The computation time T_{comp} can further be expressed in terms of instruction count and processor performance:

$$T_{\text{comp}} = \frac{N_{\text{instr}} \cdot \text{CPI}}{f_{\text{clk}}} \tag{2.2}$$

where:

- N_{instr} is the number of executed instructions,
- CPI is the average cycles per instruction,
- f_{clk} is the processor clock frequency.

Similarly, the interconnect and memory latency terms can be approximated as:

$$T_{\text{bus}} = N_{\text{mem}} \cdot L_{\text{bus}}, \quad T_{\text{mem}} = N_{\text{mem}} \cdot L_{\text{mem}} \tag{2.3}$$

where:

- N_{mem} is the number of memory accesses,
- L_{bus} is the average latency per access through the interconnect,
- L_{mem} is the average main memory latency per access.

Combining the above expressions, the total completion time can be written as:

$$T_{\text{total}} = \frac{N_{\text{instr}} \cdot \text{CPI}}{f_{\text{clk}}} + N_{\text{mem}} \cdot (L_{\text{bus}} + L_{\text{mem}}) \tag{2.4}$$

This simple model demonstrates how system-level performance can be decomposed into measurable components, which can then be obtained using performance counters in hardware or estimated through simulation.

3

Functional Verification on FPGA

Verifying register-transfer level (RTL) correctness follows a staged process, progressing from cycle-accurate simulation to FPGA-based validation only once earlier stages are complete. RTL should first be subjected to cycle-accurate simulation to verify the functionality of each block; this gives maximum signal visibility for debugging. Only once block-level and interface-level simulations are complete for all blocks intended for FPGA verification should one proceed to FPGA. For sufficiently small SoC designs, simulating the whole SoC is also possible. This provides a way to load small software onto the memory model, revealing software–hardware interactions rather than relying solely on RTL testbenches. This HW-SW co-simulation is, however, still constrained to small software sizes, and dumping signals for the whole SoC can take a matter of hours for a single simulation.

This is where FPGAs come in: they provide a way to run full-size, representative benchmarks on the SoC and test scenarios that are infeasible in simulation. This, however, comes with its own challenges. RTL written for ASIC may not always be directly compatible with FPGA; the RTL may therefore have to be modified to target FPGA while preserving functional correctness. Beyond this, a supporting toolchain is needed to synthesise and implement the design on the target FPGA, and various debug tools and analysers are needed to support debugging. This chapter covers these topics needed for functional verification on FPGA.

3.1 RTL for FPGA

ASIC-style RTL is written with a specific technology library in mind: the set of standard cells that will ultimately be used to synthesise the design into silicon. This library typically spans a wide range of cells, from primitive logic gates (AND, OR, NAND) and latches, through to more complex elements such as full adders and dedicated memory compiler IP, each characterised for the target fabrication process [20]. This is fundamentally different from how an FPGA implements logic. Rather than a library of fixed-function standard cells, an FPGA is built from an array of configurable logic blocks (CLBs) connected by a programmable interconnect fabric [17]. Each CLB is itself composed of look-up tables (LUTs), flip-flops (FFs), dedicated carry/adder logic, and multiplexers, arranged so that essentially any Boolean function can be implemented within a single CLB. In addition to CLBs, modern FPGAs provide a set of dedicated hard primitives distinct from general fabric logic: block RAM (BRAM) for on-chip memory, Digital Signal Processor (DSP)

slices for multiply-accumulate and arithmetic operations, clock management tiles (CMTs) for clock generation and distribution, and I/O blocks for interfacing with the outside world (see Figure 3.1).

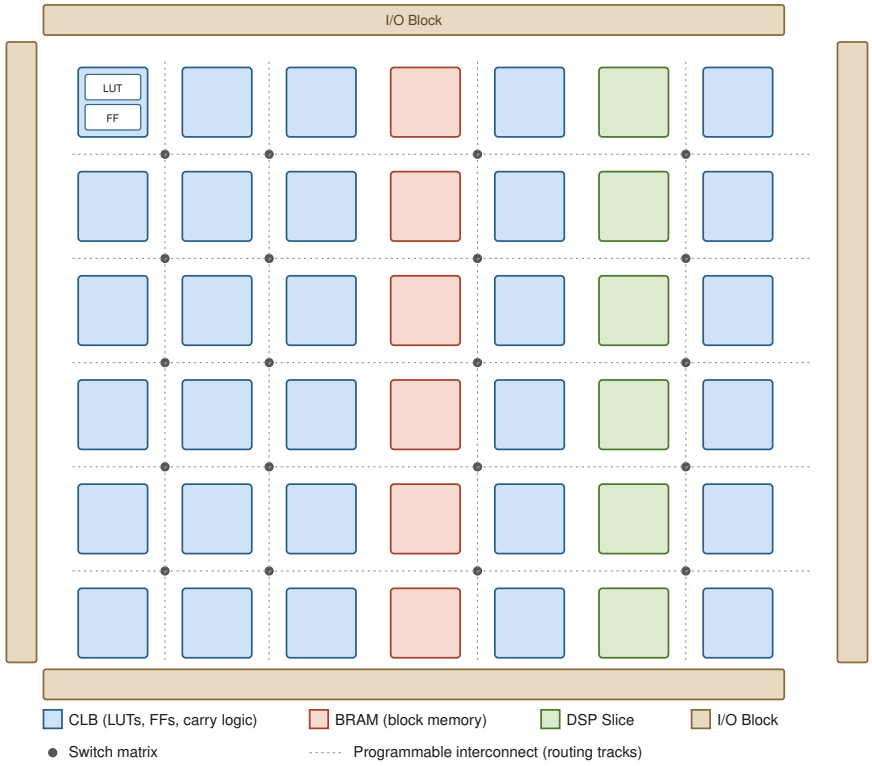

Figure 3.1: FPGA fabric architecture

Example: Clock Gating

Clock gating is a common ASIC low-power technique, in which an integrated clock gating (ICG) cell, a dedicated standard cell combining a latch and an AND gate, is used to disable the clock to a block of logic when it is not in use, reducing dynamic power consumption in the clock tree. FPGA fabric has no equivalent ICG standard cell, and directly gating a clock signal with ordinary combinational logic is generally unsafe on FPGA: dedicated global clock buffers are the intended path for clock distribution, and inserting arbitrary logic into that path risks glitches propagating into the clock network [20]. The FPGA-equivalent approach is instead to retain the flip-flop's clock input on the dedicated clock buffer and use the flip-flop's synchronous clock-enable (CE) input to hold its state, combined with a dedicated clock buffer primitive that supports glitch-free enabling (e.g. Xilinx's `BUFGCE`) where the clock network itself should also be disabled for power saving. RTL originally written with an explicit ASIC-style clock gate must therefore be restructured to express this behaviour as a clock-enable signal on the destination registers, rather than as a gated clock net.

Table 3.1 summarises further common adaptations required when porting ASIC-style RTL to an FPGA target, beyond clock gating.

Table 3.1: Common RTL adaptations required when porting ASIC-style RTL to FPGA

ASIC-style construct	Issue on FPGA	FPGA-style adaptation
Asynchronous reset distributed via a dedicated reset tree	FPGA reset routing is not optimised for large asynchronous fan-out in the same way as an ASIC reset tree, and can cause the timing violations	Synchronous reset, or an asynchronous-assert/synchronous-deassert reset synchroniser
Technology-specific memory compiler IP (SRAM macros)	Not available on FPGA; the underlying physical memory differs entirely from ASIC SRAM macros	Memory described in an inferable coding style, mapped by the synthesis tool onto BRAM primitives
Technology-specific multiplier/arithmetic IP	Not available on FPGA	Arithmetic described in an inferable coding style, mapped onto DSP slice primitives
Internal tri-state buses	Many FPGA fabrics do not support internal tri-state logic outside of dedicated I/O blocks	Multiplexer-based bus structures

3.2 Debug Tools

Once a design moves from simulation to a physical FPGA, the level of visibility into internal state drops sharply compared to what a simulation waveform provides, making dedicated debug tooling necessary to locate and diagnose issues that only manifest once hardware is running at speed [21]. FPGA prototyping debug approaches fall broadly into three categories: embedded on-chip debug cores, external test equipment, and communication-channel-based software debug.

Embedded Logic Analysers

The major FPGA vendors provide embedded logic analyser IP that is instantiated directly within the target design: Xilinx's ILA and Intel/Altera's SignalTap are the most widely used examples [21]. These cores use FPGA logic resources to implement triggering and FPGA memory blocks to buffer captured samples, with the captured data read out over the existing Joint Test Action Group (JTAG) connection rather than requiring dedicated debug pins [22]. Because sample storage is limited to on-chip memory, the depth of capture achievable this way is considerably shallower than what an external logic analyser could record, and the core itself consumes device resources that are otherwise unavailable to the design under test, typically recommended to be kept to a small fraction of overall device capacity [21]. A complementary core, Virtual I/O (VIO), allows internal signals to be both observed and driven live over the same JTAG connection, without requiring the signal to be brought out to a physical pin, which is useful for exercising internal control or

configuration logic directly from a host machine during debug.

External Test Equipment

Alternatively, internal signals can be routed to dedicated I/O pins and observed with an external oscilloscope or logic analyser. This avoids consuming internal FPGA logic and memory resources for debug purposes, and offers substantially greater sample depth and triggering flexibility than an embedded core constrained to on-chip memory [21]. The trade-off is a requirement for spare I/O pins, which may not be available on a pin-constrained design, and the need to identify signals of interest and route them to pins in advance, since, unlike an embedded core, changing which signals are observed typically requires a full re-synthesis and reprogramming of the device.

Software-Level Debug via JTAG

Where the design under test includes a processor core, a further debug approach operates at the software level rather than the raw signal level: a JTAG-connected debug module exposes the processor's internal state (registers, memory, execution control) to an external debugger, allowing software running on the core to be halted, single-stepped, and inspected using standard tools such as the GNU Debugger (GDB). For RISC-V cores specifically, this is standardised by the RISC-V External Debug Support specification [23], which defines a common debug module architecture and distinguishes between *halt-mode* debugging, where the debugger halts the core and inspects state while execution is paused, and *run-mode* debugging, where a software debug agent on the core communicates with the debugger without halting execution, relevant for components with real-time constraints that cannot tolerate being stopped [23]. This approach provides visibility into software and architectural state rather than arbitrary internal RTL signals, and is therefore complementary to, rather than a replacement for, the signal-level tools described above.

Choosing Between Approaches

These approaches are not mutually exclusive, and a typical FPGA prototyping flow draws on more than one depending on the nature of the problem being debugged. Embedded logic analysers are well suited to observing internal RTL signals without consuming spare I/O, at the cost of device resources and capture depth; external test equipment offers greater capture depth and flexibility at the cost of pin availability and debug-iteration speed; and software-level JTAG debugging is suited to diagnosing problems in the software or architectural state of an embedded core, rather than in the surrounding RTL fabric.

Open-Source Debug Tools

Vendor-provided debug tooling is proprietary and tied to a specific FPGA vendor's toolchain, which is not always desirable for an open-source design flow. A number of open-source alternatives exist covering the same debug categories described above.

For JTAG-based software debug, OpenOCD (Open On-Chip Debugger) provides an open-source implementation of a JTAG-connected debug interface, originally developed for ARM cores and since extended to a wide range of target architectures [24]. OpenOCD exposes a GDB server over its JTAG connection, allowing standard GDB to be used for source-level debugging of a target core without relying on vendor-specific debug software, and is the de facto standard open-source bridge between GDB and RISC-V debug modules implementing the specification described above.

For embedded logic analysis, LiteScope, part of the open-source LiteX SoC ecosystem, provides a vendor-agnostic alternative to ILA and SignalTap [25]. Like its vendor equivalents, LiteScope instantiates a configurable logic analyser core directly in the FPGA fabric, buffering captured samples in block RAM with configurable triggering; unlike the vendor tools, captured data can be streamed off-chip over a UART, Ethernet, or PCI Express (PCIe) bridge rather than only through vendor-specific JTAG software, and results can be exported to standard open formats such as Value Change Dump (VCD) for inspection in an open-source waveform viewer such as GTKWave, rather than being tied to a single vendor's analysis graphical user interface (GUI).

3.3 Proposed Functional Verification Methodology

Building on the RTL adaptation considerations of Section 3.1 and the debug tooling introduced in Section 3.2, this section proposes a structured methodology for the functional verification of newly introduced logic on open-source SoC platforms within an FPGA-based PoC environment. The methodology is designed to operate under the reduced observability inherent to FPGA prototyping relative to simulation, while still providing confidence that a design is correct before it is relied upon for subsequent performance evaluation.

The staged, simulation-then-FPGA structure of this methodology is not itself novel: it reflects standard practice in both industrial verification flows and prior FPGA-based verification work [7, 8, 9]. What distinguishes the methodology proposed here is how verification effort is allocated once that staged structure is in place, specifically tailored to the constraints of resource-limited, open-source SoC prototyping rather than to a fully resourced commercial verification effort. Existing FPGA-based verification work surveyed in Chapter 1 either depends on vendor-specific debug tooling [9], remains tightly coupled to a single target application [11], or documents the practical challenges of FPGA prototyping without proposing a structured methodology for managing them [10]. In contrast, the methodology below makes three concrete departures from these flows and from conventional block-level verification practice such as UVM-based simulation. First, rather than re-verifying an entire platform's constituent IP block by block, the existing open-source platform is treated as a pre-verified baseline, and verification effort is concentrated exclusively on newly introduced logic, added and verified incrementally and in isolation. Second, root-causing simulation-to-FPGA discrepancies draws explicitly on the open-source

platform's public issue tracker as a structured diagnostic step, an option unavailable to closed, vendor-maintained flows. Third, since exhaustive simulation-based coverage metrics are impractical to measure once verification has moved to FPGA, functional coverage is instead established through paired nominal and negative testing of newly introduced detection logic, substituting for the coverage tooling assumed by conventional UVM-based flows.

Simulation-First, Staged Verification

The central principle of the proposed methodology is that FPGA-based verification is not the starting point for establishing correctness, but the final stage of a staged process. Cycle-accurate simulation is used first, since it offers maximum signal visibility and allows obvious bugs to be identified and corrected without the overhead of synthesis and bitstream generation. Only once a design has passed simulation-level verification should it be carried forward to FPGA, at which point simulation results can serve as the reference against which FPGA behaviour is checked, rather than correctness being established from scratch on hardware.

Baseline Platform Verification

Where an existing open-source platform is used as a starting point, its constituent IP is not re-verified exhaustively block by block, since not every block provided by the platform will necessarily be exercised by the system under evaluation. Instead, a minimal test exercising only the components essential to basic operation, typically the processor core, memory subsystem, interconnect, and clock/reset generation, is used to confirm that the unmodified baseline platform functions correctly before any new logic is introduced. This establishes a known-good starting point against which the effect of subsequent modifications can be isolated.

Incremental Verification of Newly Introduced Logic

Rather than integrating all new logic at once, each new hardware block introduced to the platform, such as additional instrumentation or an integrated accelerator, is verified incrementally and in isolation before being combined with other new logic. For each addition, software is written to exercise the new functionality and is first run in simulation, where the result is checked against a manually derived or reference-model expected value. Only once this simulation-level result is confirmed correct is the same test carried forward to FPGA and checked for agreement with the simulation result. This incremental approach ensures that, should a discrepancy arise between simulation and FPGA behaviour, it can be attributed to a specific, recently introduced piece of logic rather than to an unknown combination of several simultaneous changes.

Root-Cause Analysis Using Open-Source Resources

Where FPGA behaviour diverges from a simulation-verified expectation despite the design being logically correct in simulation, the discrepancy is typically attributable

to a synthesis-level or FPGA-specific effect rather than a functional error in the RTL itself, for example a construct that behaves correctly in simulation but does not synthesise as intended onto FPGA fabric. For designs built on an open-source platform, a public issue tracker associated with the platform is a valuable first point of reference in diagnosing such discrepancies, since related synthesis-specific issues may already have been reported and resolved by the platform's wider community, potentially narrowing down a root cause considerably faster than debugging from first principles alone.

Functional Coverage through Negative Testing

Confirming that newly introduced logic behaves correctly under nominal conditions is not sufficient to establish full functional coverage; logic intended to detect a specific condition, such as a contention or stall detection mechanism, should additionally be verified under a deliberately constructed negative test that induces the condition it is meant to detect, in addition to confirming its expected, non-triggering behaviour under normal operation.

Together, these stages form a top-down, incremental verification flow: a known-good baseline is established first, new logic is added and verified one piece at a time in simulation before being promoted to FPGA, discrepancies between the two are root-caused with the aid of available open-source resources where applicable, and functional coverage is confirmed through both nominal and negative testing of newly introduced detection logic.

4

SoC Performance Modelling and Evaluation

As discussed in Section 2.4, performance models are essential for evaluating the metrics of interest in a given SoC. In the context of FPGA-based prototyping, however, the constraints of limited resources and reduced observability—as described in Section 3.2—restrict the level of detail that can be captured in such models. In practice, it is not feasible to implement hardware counters for every parameter simultaneously. Therefore, depending on the specific goals of the performance evaluation, only selected components of the system are analysed.

To this end, this chapter presents a set of performance models from the literature that are relevant to SoC systems, and develops a methodology for evaluating these models on FPGA prototypes.

4.1 Benchmarks

Any SoC architecture is typically designed to target a *workload space*, which represents the set of all possible applications that the system is expected to execute. From this broader space, a *reference workload* is defined as a subset containing benchmarks that are representative of the overall workload characteristics. The selection of this reference workload is largely dependent on the designer's judgement and domain expertise.

In certain application domains, such as embedded systems, the workload space may be sufficiently constrained such that the reference workload can effectively encompass the entire set of relevant applications. In contrast, for domains such as general-purpose computing, the workload space is significantly larger and more diverse. As a result, even the reference workload may remain extensive and require further refinement. The objective of benchmark selection, therefore, is to reduce the reference workload into a manageable subset, often referred to as a *reduced workload*, while preserving its representativeness [18].

As discussed in Section 2.4, several standardised benchmark suites have been developed for different application domains to facilitate systematic system evaluation. However, in practice—particularly under tight development timelines or during early-stage exploration—it may not be feasible to execute all benchmarks within the reference workload. Consequently, various techniques have been proposed to reduce the workload while retaining its essential characteristics.

Statistical methods such as Principal Component Analysis (PCA) and the Plackett–Burman design are commonly used for this purpose [18]. These approaches identify and eliminate redundant benchmarks that exhibit similar behavioural characteristics or stress the same aspects of the system. By selecting a representative subset of benchmarks, these methods enable designers to significantly reduce evaluation time while maintaining meaningful performance insights.

These methods are explicitly designed to produce microarchitecture-independent reduced sets. They cluster benchmarks by workload characteristics, such as instruction mix and cache or branch behaviour, rather than by measured performance on one specific target. This means the resulting subset need not be re-derived for every new design.

This independence claim has been tested against conventional microarchitectural enhancements, such as larger reorder buffers, deeper caches, and prefetching. PCA and Plackett–Burman-based subsets retained good relative accuracy across these configurations [26]. However, this testing did not cover offloading to a dedicated accelerator. An accelerator changes which code path executes at all, not just how efficiently a fixed code path runs. A benchmark subset validated for a scalar core may therefore not remain representative once an accelerator is introduced. Benchmark selection should accordingly be re-validated against the specific SoC configuration under evaluation, particularly where accelerators are involved.

4.2 Performance Metrics

Not every metric of interest to a final ASIC implementation can be measured meaningfully on an FPGA-based PoC. This section categorises performance metrics relevant to SoC evaluation into three groups: those both relevant and reliably measurable on FPGA, those relevant to the eventual ASIC target but impractical or misleading to measure on FPGA, and those that are FPGA-specific, useful only for PoC-level debugging rather than as an indicator of final-silicon behaviour.

4.2.1 Measurable on FPGA and ASIC relevant

Metrics that characterise the architectural behaviour of a design can translate well from FPGA to ASIC, provided the parameter measured is fixed by an architectural protocol contract rather than exposed directly by the underlying implementation fabric, as discussed in Chapter 1. Where such a contract holds, these metrics are counted in cycles or events rather than absolute physical units, and the relative behaviour they capture (which access pattern causes contention, which component is a bottleneck) is expected to hold regardless of whether the design is ultimately realised on FPGA or ASIC. This class includes:

- **Execution time / cycle count**, obtained directly from cycle counters, and the resulting derived metrics such as instructions per cycle (IPC) and speedup between configurations.
- **Cache/memory hit and miss rates**, and associated stall cycles, which reflect the algorithmic memory access pattern of the workload rather than the

physical memory technology used to implement it, provided the memory interface's access latency is itself protocol-guaranteed rather than fabric-dependent.

- **Interconnect and memory bandwidth utilisation**, capturing contention and throughput at the system level.
- **Functional throughput**, such as operations completed per unit time (measured in cycles) for a given workload.

Where no such protocol contract exists, and a counted value instead reflects how long the underlying physical implementation actually takes to complete an operation, this portability does not hold: the counted value can diverge between FPGA and ASIC targets, since it depends on the specific memory or datapath primitives instantiated on each target [5].

4.2.2 Measurable on FPGA but Inaccurate for ASIC

Several metrics that matter for a final ASIC implementation cannot be measured accurately on FPGA, not because they are unmeasurable in a strict sense, but because the underlying implementation fabric differs sufficiently from a standard-cell ASIC that the measured values are not representative. An empirical quantification of this gap can be seen in [5]: for logic implemented using only programmable look-up tables and flip-flops, FPGA designs require on average 35 times more silicon area than an equivalent standard-cell ASIC implementation, exhibit roughly 3–4 times greater critical path delay, and consume approximately 12 times more dynamic power. These gaps narrow, but do not close, when a design makes use of dedicated hard blocks such as DSP slices and block memories rather than relying purely on programmable fabric.

This has direct implications for which metrics can be trusted from an FPGA measurement:

- **Maximum clock frequency (F_{max})**. FPGA timing is dominated by the delay of the programmable interconnect fabric, which differs fundamentally from the standard-cell timing that will determine the ASIC's actual achievable frequency; an FPGA-measured F_{max} is not a reliable predictor of ASIC F_{max}.
- **Silicon area**. FPGA resource utilisation is reported in terms of LUTs, flip-flops, and hard block instances, which does not map linearly, or even consistently, onto the silicon area an equivalent ASIC implementation would occupy.
- **Power consumption**. FPGA power includes substantial static and reconfiguration related overhead intrinsic to the programmable fabric that has no ASIC counterpart, making absolute FPGA-measured power figures unrepresentative of ASIC power, even where the relative power impact of an architectural change may still offer some directional insight.

For these metrics, an FPGA PoC can still support qualitative, relative comparisons between design alternatives implemented on the same FPGA, but absolute values should not be taken as predictive of final ASIC characteristics.

4.2.3 FPGA-Specific Metrics

A final category of metrics has no ASIC equivalent at all, and exists purely to support the FPGA-based development and debugging process itself, distinct from evaluating the design under test. These include:

- **FPGA resource utilisation as a percentage of device capacity** (LUT, FF, BRAM, DSP, I/O), used to judge whether a design fits a candidate device and how much headroom remains
- **Synthesis, implementation, and bitstream generation time**, relevant to development iteration speed on FPGA but not a property of the design's runtime behaviour.
- **Routing congestion and timing closure margin** reported by FPGA implementation tools, which relate to whether a given design successfully maps onto the specific target device rather than to the design's intrinsic performance.
- **Debug-interface-specific overhead**, such as the resources consumed by embedded logic analyzer cores or JTAG-based debug infrastructure, which exist solely to support the PoC development process and are not part of the design being evaluated.

These metrics are essential to successfully bringing up and debugging a design on FPGA, but should not be conflated with the performance metrics of the first two categories, since they describe the prototyping process rather than the system under test.

4.3 Performance Models

A performance model is preferred over a simple counter that directly measures the final metric of interest because of the difference between a black-box and a white-box approach. A single counter recording, for example, total execution cycles reports *that* a given configuration performs a certain way, but offers no insight into *why*, since the underlying contributors to that number remain opaque. A performance model, by contrast, decomposes the metric into its constituent architectural parameters, providing a white-box view that identifies which specific component or effect is responsible for the observed behaviour, and therefore which parameter should be targeted to improve it. This distinction motivates the parameter-level, white-box models discussed in this section.

4.3.1 Single Core Performance Models

Single-core processors are typically employed in lightweight systems, such as embedded environments. These processors can be broadly classified into in-order and out-of-order architectures. In-order cores execute instructions strictly in program order, which can lead to pipeline stalls in the presence of data dependencies. In contrast, out-of-order cores allow instructions to be executed non-sequentially, enabling the processor to exploit instruction-level parallelism. As a result, data-independent instructions can proceed without stalling the pipeline, improving overall performance.

The performance model for a scalar, in-order pipeline considered in this work is adopted from [27]. The execution time of a workload is given by Equation 4.1, with the corresponding parameters listed in Table 4.1.

$$T = N + m_{Icache} \cdot \ell_{MEM} + m_{Dcache} \cdot \ell_{MEM} + N_{br}^{T} + m_{br} \cdot d + N_{ld\_deps} + N_{ld/st} \cdot (\ell_{Dcache\_hit\_time} - 1) \tag{4.1}$$

Table 4.1: Single-core in-order scalar model parameters

Parameter	Description
N	number of dynamically executed instructions
m_{Icache}	number of I-cache misses
m_{Dcache}	number of D-cache misses
ℓ_{MEM}	memory access time in cycles
N_{br}^{T}	number of taken branches
m_{br}	number of branch mispredictions
d	pipeline depth between fetch and execute
$N_{ld\ deps}$	number of instructions that depend on previous load
$N_{ld/st}$	number of loads and stores
$\ell_{Dcache\ hit\ time}$	D-cache hit time

As described in [27], the flow of instructions through the pipeline is disrupted by miss events and pipeline hazards. Miss events include penalties arising from instruction and data cache misses, as well as translation lookaside buffer (TLB) misses, and the additional cycles required to refill the front-end pipeline following a branch misprediction. While forwarding logic mitigates many data dependency stalls, load instructions still incur a penalty corresponding to the data cache hit latency. For formula simplification, the model ignores TLB misses, treating them the same as cache misses.

The model captures first-order performance effects by assuming that miss events and hazards occur independently. Although this assumption simplifies the analysis, it has been shown to provide sufficiently accurate estimates for practical purposes.

Modern processor cores typically provide hardware performance counters that expose basic metrics such as the number of retired instructions (N) and total cycle count. Additional event counters may be configured to capture other parameters in the model, although their availability is often implementation-dependent.

In addition to the scalar in-order model presented here, more advanced models exist for in-order superscalar processors (with issue width greater than one) and out-of-order processors. These models account for additional sources of parallelism and complexity, and are discussed in [28] and [29], respectively.

4.3.2 Multicore Performance Models

The primary motivation for transitioning to multicore systems is to exploit parallelism in workloads. This parallelism may manifest either as multiprogram or multithreaded execution. Multithreaded workloads consist of multiple threads belonging

to the same application, typically sharing data and operating on common address spaces. In contrast, multiprogram workloads involve the concurrent execution of independent programs, which may not share data.

From an architectural perspective, the key distinction between single-core and multi-core processors lies in the presence of multiple cores that may share system resources. For instance, a quad-core processor may employ private L1 caches for each core while sharing a unified L2 cache among all cores. Such resource-sharing mechanisms are common in both multiprogram and multithreaded systems.

While the performance model of an individual core remains fundamentally similar to the single-core case discussed in Section 4.3.1, the overall system performance is affected by interactions between cores. These interactions arise due to contention for shared resources such as caches, memory bandwidth, and interconnects, and must therefore be accounted for when modelling multicore system performance.

Evaluating multicore systems running multiple applications requires metrics that capture both user-perceived performance and overall system efficiency. In such environments, co-running programs interfere with each other due to shared resources, making single-program metrics like cycles per instruction (CPI) insufficient. Two widely used metrics that reflect these perspectives are Average Normalised Turnaround Time (ANTT) and System Throughput (STP) [30].

ANTT is a user-oriented metric that quantifies the average slowdown experienced by applications when executed concurrently, relative to when they run alone. It is defined as the arithmetic mean of normalised execution times:

$$\text{ANTT} = \frac{1}{n} \sum_{i=1}^{n} \frac{C_i^{MP}}{C_i^{SP}}, \tag{4.2}$$

where C_i^{SP} and C_i^{MP} denote the execution time of program i in isolation and in a multiprogram setting, respectively. Lower ANTT indicates better user-perceived performance.

In contrast, STP is a system-oriented metric that measures how efficiently the system makes progress across all programs. It is defined as the sum of normalised speedups:

$$\text{STP} = \sum_{i=1}^{n} \frac{C_i^{SP}}{C_i^{MP}}. \tag{4.3}$$

Higher STP reflects better utilisation of system resources and improved aggregate throughput.

In both equations, n is the number of independent co-executing programs. Together, ANTT and STP provide complementary insights, capturing the trade-off between fairness to individual applications and overall system productivity.

4.3.3 Roofline Model

The Roofline model provides an intuitive, architecture-aware framework for analysing application performance by relating computation to data movement. It is motivated

by the observation that performance on multicore systems is typically limited either by peak compute capability or by memory bandwidth, depending on how effectively a program reuses data [31].

The central parameter in the model is the *operational intensity* (OI), defined as the ratio of useful operations to data movement from main memory:

$$\text{OI} = \frac{\text{Total floating-point operations}}{\text{Total data movement (bytes)}} \tag{4.4}$$

Using this quantity, the attainable performance is bounded by:

$$P = \min\left(P_{\text{peak}},\ \text{OI} \times B_{\text{peak}}\right) \tag{4.5}$$

where P_{peak} denotes peak computational throughput in GFlops/s and B_{peak} the peak memory bandwidth in Gbytes/s.

As illustrated in Figure 4.1, performance increases linearly with operational intensity in the bandwidth-bound region, following the slope determined by memory bandwidth. Beyond the ridge point (the intersection of the two bounds), performance saturates at the compute peak, and the execution becomes compute-bound. This distinction provides clear guidance for optimisation: improving data locality or data movement speed shifts execution to the right (increasing OI), while exploiting parallelism and vectorisation raises the achievable compute ceiling.

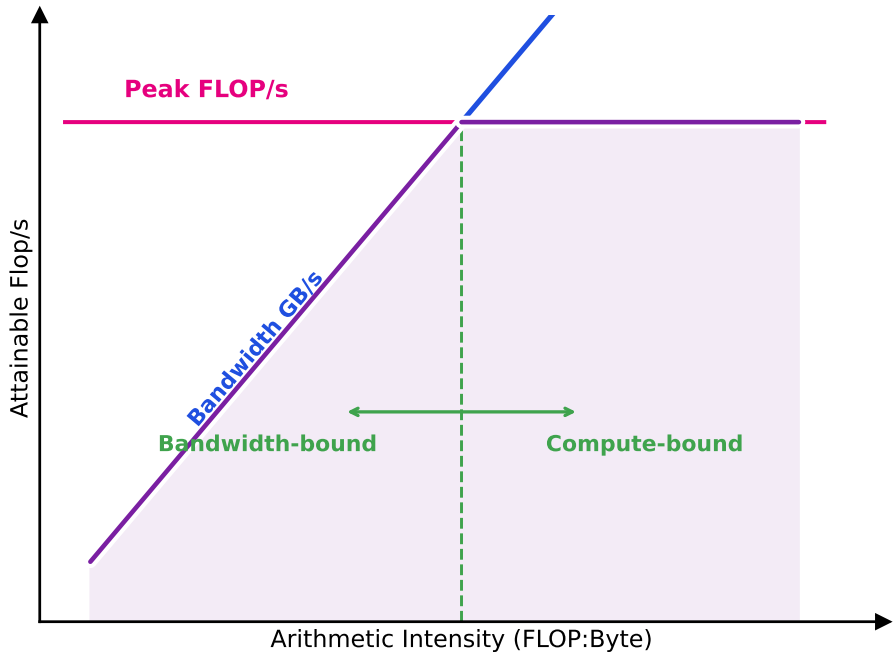

Figure 4.1: Roofline model

4.3.4 Accelerator-Based SoC Performance Models

Accelerators offload specialised tasks from the processor and can deliver higher performance by exploiting hardware specialisation. However, their effectiveness depends not only on computational capability but also on communication overheads

and data movement costs. The LogCA model provides a framework to analyse these trade-offs [32].

The LogCA model is a high-level analytical framework for reasoning about the performance of hardware accelerators in heterogeneous systems. It is motivated by the observation that accelerator performance depends not only on computational speed, but also on the overhead and latency incurred when offloading work from the host processor. In many cases, these communication costs can dominate execution time, especially for small problem sizes, leading to limited or even negative speedup.

LogCA abstracts the system into three components: a host processor, an accelerator, and an interface connecting them. The model characterises performance using five key parameters summarised in Table 4.2.

Table 4.2: LogCA model parameters.

Parameter	**Symbol**	**Description**
Latency	L	Time to transfer data across the interface
Overhead	o	Host-side setup cost for offloading work
Granularity	g	Size of the offloaded data
Computational index	C	Compute cost per unit of data (cycles/byte)
Acceleration	A	Relative speedup of accelerator vs. host

The execution time on the host (without acceleration) is modelled as:

$$T_0(g) = C \cdot f(g), \tag{4.6}$$

where $f(g)$ represents the algorithmic complexity (e.g., g^β). When using an accelerator, execution time becomes:

$$T_1(g) = o + L + \frac{C \cdot f(g)}{A}. \tag{4.7}$$

The resulting speedup is:

$$\text{Speedup}(g) = \frac{T_0(g)}{T_1(g)} = \frac{C \cdot f(g)}{o + L + \frac{C \cdot f(g)}{A}}. \tag{4.8}$$

As illustrated in Figure 4.2, the model captures two key regimes. For small granularity, performance is dominated by overhead (o) and latency (L), resulting in low or no speedup. As granularity increases, these fixed costs are amortised, and speedup improves, eventually approaching the peak acceleration A. The model also defines a break-even point (g_1), which represents the minimum data size required for the accelerator to outperform the host.

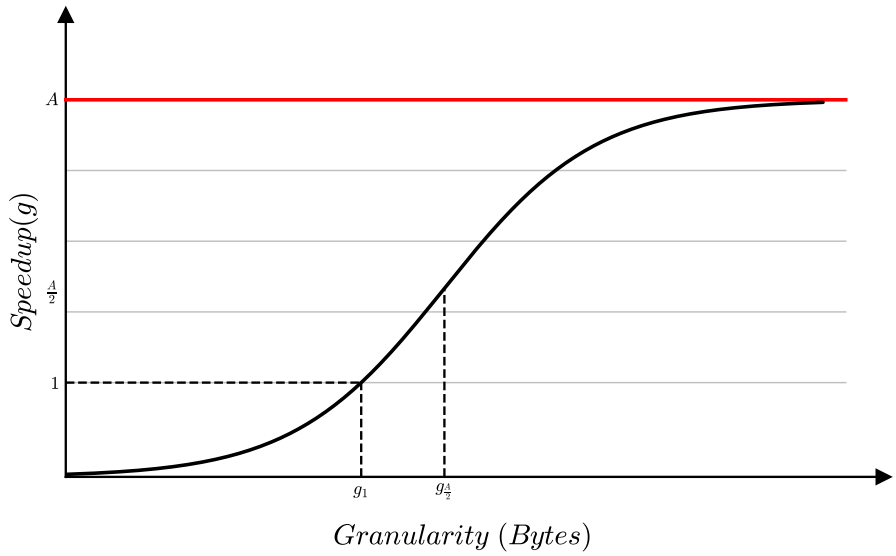

Figure 4.2: Graphical representation of how acceleration speedup is affected by granularity

Parameter Estimation. LogCA parameters can be obtained through a combination of profiling, microbenchmarks (small, targeted test programs that isolate and measure a single system property rather than executing a full application, e.g., a tight loop that repeatedly issues a single data transfer to measure interconnect latency in isolation), and analytical estimation:

- **Computational index (C):** Measured by profiling the execution time of the application on the host across varying input sizes, and fitting a model $T_0(g) = C \cdot f(g)$ using regression analysis.
- **Overhead (o):** Estimated from execution time at very small granularities, where computation is negligible and the total time is dominated by setup overhead.
- **Latency (L):** Determined using microbenchmarks or derived from peak bandwidth (e.g., $L \approx 1/\mathrm{BW}_{\mathrm{peak}}$), depending on whether the accelerator is on-chip or off-chip.
- **Acceleration (A):** Measured as the asymptotic speedup at large granularities, or approximated from peak throughput ratios between accelerator and host.
- **Granularity (g):** Defined by the input size or amount of data offloaded to the accelerator.

This parametrisation enables LogCA to provide early-stage performance insights without requiring detailed architectural models. In particular, it highlights key design trade-offs between improving compute capability (A) and reducing communication costs (o, L), making it well-suited for evaluating accelerator-based SoC designs.

4.4 Proposed Performance Evaluation Methodology

Based on the performance models discussed in the previous sections, this work proposes a structured methodology for performance monitoring of open-source SoC platforms on FPGA-based PoC systems. The methodology is designed to operate under the constraints of limited FPGA resources and restricted observability, while still enabling meaningful performance evaluation.

Benchmark Selection

The first step in the methodology is the selection of appropriate benchmarks. The choice of benchmarks depends on the intended application domain of the SoC under design. For example, general-purpose processors may be evaluated using benchmark suites such as SPEC CPU, while machine learning or embedded systems may require domain-specific workloads such as MLPerf or ULPMark. The selected benchmarks should be representative of real-world workloads to ensure meaningful performance evaluation.

Baseline System Definition

A baseline system configuration is established to serve as a reference point for performance evaluation. This includes defining the architectural configuration of the SoC, such as the number of cores, cache hierarchy, interconnect type, and any integrated accelerators. Performance measurements are first obtained on this baseline system before any modifications or optimisations are introduced.

Performance Metric and Model Selection

Performance models are derived for different components of the system to enable component-level analysis. The primary components considered include the processor cores, memory subsystem, and interconnect (bus or NoC). Secondary components such as accelerators and additional buses may also be modelled depending on the evaluation goals. Peripherals are generally excluded from detailed performance modelling, as they are typically standardised and do not constitute primary performance bottlenecks.

Two levels of abstraction are considered in the modelling process:

- **Abstract (high-level) models:** These are used for system-level estimation and comparison. Key metrics include:
  - Throughput, which represents a component's operations per unit time and the actual achieved performance. Instructions per cycle (IPC) of a processor core is an example.
  - Bandwidth, which refers to the maximum data rate that can be transferred and represents the theoretical maximum capability of the component.

  - Latency, which measures performance from the perspective of a single operation rather than overall system performance. This can be more relevant for latency-sensitive workloads such as real-time systems.
  - Arithmetic intensity, particularly for processor and accelerator workloads
- **Fine-grained models:** These are used to identify bottlenecks and guide optimisation. They capture detailed microarchitectural effects such as cache behaviour, pipeline stalls, and interconnect congestion.

Performance Counter Identification and Mapping

Modern SoCs, particularly those based on RISC-V, provide hardware performance counters that can be accessed through control and status registers (CSRs). These counters are used to capture events corresponding to parameters in the performance models. Depending on the chosen open-source SoC platform, the required performance counters may need to be implemented if they are not available.

Typical counters include:

- **Processor counters:** cycle count, instructions retired, stall cycles, branch events
- **Memory system counters:** cache hits and misses, load/store wait cycles
- **Interconnect counters:** bus accesses, arbitration stalls
- **NoC counters:** flit or byte counts, packet latency (minimum, maximum, average), hop count, buffer occupancy, injection rate

A key challenge in FPGA-based performance monitoring is the mismatch between model parameters and available hardware counters. It is typically not feasible to measure all parameters simultaneously due to resource constraints.

To address this, a top-down approach is adopted:

- Initial measurements are performed using high-level metrics to estimate overall system performance
- Based on these results, critical components and bottlenecks are identified
- Fine-grained instrumentation is then selectively enabled for these components in subsequent runs

This approach minimises FPGA resource utilisation while ensuring that detailed analysis is performed only where necessary.

Performance Counter Access Mechanisms

The collected performance data must be accessible for analysis. Several mechanisms are considered:

- **Software-accessible registers:**
  - Memory-mapped I/O (MMIO) registers
  - Core-local CSRs

- **Debug interfaces:**
  - JTAG or external debug modules
- **Streaming mechanisms:**
  - Direct memory access (DMA) to stream performance data to memory

While DMA-based approaches enable continuous data collection, they may introduce additional load on the interconnect and must therefore be used carefully to avoid perturbing system performance.

Limitations and Trade-Offs

The methodology (Section 4.4) follows an iterative process in which the performance of the system and its individual components is evaluated using different metrics. For instance, measuring overall system throughput may require a different set of performance counters than those used to evaluate NoC throughput. Given the constraints of FPGA-based prototyping, a trade-off must be maintained between observability and resource usage. Instrumentation may be selectively enabled, disabled, or rotated across FPGA builds to balance these constraints while maintaining relevance to ASIC implementations.

5

Case Study Methodology

This chapter presents the case study used to evaluate the proposed functional verification and performance monitoring methodologies described in the previous chapters. The objective of the case study is to demonstrate how these methodologies can be applied in practice to an open-source SoC platform within an FPGA-based PoC environment and assess the soundness of the proposed approaches. The results obtained from the case study are discussed in Chapter 6.

5.1 Target System

Consider a design team that has developed a new hardware accelerator but does not yet have a complete SoC in which to test it. Building an entire SoC from scratch purely to host and evaluate this accelerator would be costly and time-consuming. Instead, the team selects an existing open-source SoC platform and a benchmark representative of their target workload, and integrates the new accelerator into this platform. This produces two systems to evaluate: a baseline system, consisting of the unmodified open-source platform, and an accelerator-augmented system, consisting of the same platform extended with the newly integrated accelerator. Both systems are first subjected to functional verification to confirm the correctness of the implemented system, including the performance monitoring hardware counters added for evaluation. Each system is then subjected to performance evaluation using models adapted to its architecture. Comparing the two evaluations reveals where bottlenecks occur in each system, and shows how the bottleneck shifts once the accelerator is introduced.

5.1.1 Platform Selection

For this case study, the Pulpissimo SoC platform from the PULP ecosystem is selected as the baseline system. The PULP platform provides a collection of open-source SoC implementations that are readily available for use, without requiring hardware generation frameworks as in systems such as Chipyard.

PULP-based platforms are built around the RISC-V instruction set architecture and span a wide range of configurations, including single-core, multicore, and multi-cluster systems. This flexibility makes them suitable for a variety of application domains, including embedded systems and accelerator-based computing.

Pulpissimo, in particular, is a lightweight single-core SoC with support for hardware accelerators. Its relatively simple architecture makes it well-suited for exploratory analysis in this thesis, while still capturing the essential components of a modern SoC, such as a processor core, memory subsystem, interconnect, and peripheral interfaces.

5.1.2 Baseline System - Single Core

Pulpissimo is the microcontroller-class architecture within the PULP platform, a collaborative open-source initiative between ETH Zürich and the University of Bologna.

Pulpissimo is a single-core SoC platform that integrates a processor core, memory subsystem, interconnect, and a rich set of peripherals. A block diagram of the system architecture is shown in Figure 5.1. The architecture supports two alternative RISC-V cores:

- **CV32E40P core:** An in-order, single-issue processor with a 4-stage pipeline and an IPC close to 1. It supports the RV32I base instruction set, along with extensions such as RV32C (compressed instructions) and RV32M (integer multiplication), and optionally RV32F (single-precision floating point). It is designed for energy-efficient signal processing applications.
- **Ibex core:** A lightweight in-order, single-issue processor with a 2-stage pipeline, designed for ultra-low-power and low-area applications. It supports the RV32I base instruction set and RV32C compressed instructions, with optional support for RV32M and RV32E extensions.

This case study will use the CV32E40P core.

A key feature of Pulpissimo is its autonomous I/O subsystem, implemented using a micro-DMA (uDMA) engine. The uDMA enables peripherals to transfer data directly to and from L2 memory without continuous processor involvement.

The platform includes both interleaved and non-interleaved L2 memory banks. In the case of interleaved banks, consecutive cache lines are distributed across multiple memory banks in a parallel fashion; for example, cache line 1 is placed in bank 1, cache line 2 in bank 2, and so on, eventually wrapping around to the first bank. This interleaving scheme enables concurrent access to multiple memory banks, thereby improving memory bandwidth and reducing access latency for data-parallel workloads such as vector operations. In contrast, non-interleaved memory banks store contiguous data within the same bank.

The tightly coupled data memory (TCDM) interconnect is responsible for connecting the processor core, memory banks, accelerators and the uDMA. It is implemented as a logarithmic interconnect, enabling scalable and efficient communication between multiple masters and memory banks. The TCDM interconnect also interfaces with Advanced Peripheral Bus (APB) and AXI4 bridges, enabling access to peripheral buses and external components. The APB bridge provides connectivity to low-speed peripherals, while the AXI4 bridge facilitates communication with high-bandwidth components. In this case, the AXI4 interface is used to connect the SoC to off-chip main memory, allowing data transfers between the tightly coupled memory

subsystem and external memory resources. Due to its central role in data movement and potential impact on system performance, the TCDM interconnect is selected as the primary focus of performance evaluation in this case study.

Pulpissimo supports the integration of Hardware Processing Engines (HWPEs), which act as accelerators. These accelerators share the system memory with the processor and are accessed through the memory-mapped interface, enabling efficient offloading of compute-intensive tasks such as vector operations.

Pulpissimo provides support for a variety of standard peripheral interfaces, including Serial Peripheral Interface (SPI), I2C, UART, Inter-IC Sound (I2S), camera interfaces (CPI), HyperBus, and JTAG. These peripherals are integrated with the uDMA subsystem for efficient data movement.

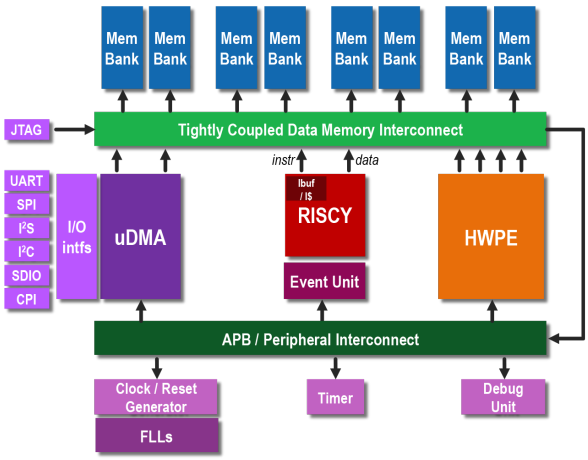

Figure 5.1: Pulpissimo SoC architecture. Figure sourced from [1].

Performance evaluation of the baseline single-core-only system follows the model of equation 4.1 (shown again below as equation 5.1).

$$T = N + m_{Icache} \cdot \ell_{MEM} + m_{Dcache} \cdot \ell_{MEM} + N_{br}^{T} + m_{br} \cdot d + N_{ld\_deps} + N_{ld/st} \cdot (\ell_{Dcache\_hit\_time} - 1) \tag{5.1}$$

This model was formulated for a cached memory hierarchy, in which instruction and data misses are penalised by a fixed memory access latency ℓ_{MEM}. Pulpissimo has no instruction or data cache: the core accesses a single-level TCDM directly, with a fixed 1-cycle access latency in the absence of contention. The model is therefore adapted to this architecture as:

$$T = N + I_{miss} + D_{stall} + N_j + N_{br}^{T} \cdot d + N_{ld\_deps} + N_{j\_deps} + (N_{ld} + N_{st}) \cdot (\ell_{TCDM} - 1) \tag{5.2}$$

Each term in this model is mapped either onto an architectural event counter internal to the core, or onto a new counter added specifically to observe contention on the TCDM interconnect. Table 5.1 lists each parameter of the adapted model, its counterpart (if any) in the original model, and whether it is unchanged, modified, or newly added.

Table 5.1: PULP adapted single-core model parameters, mapped to the original model in equation 5.1

New parameter	Description	Equivalent in original model	Status
N	Total instructions retired	N	Same
I_{miss}	Cycles stalled on instruction fetch	$m_{Icache} \cdot \ell_{MEM}$	Modified
D_{stall}	Cycles stalled on data access	$m_{Dcache} \cdot \ell_{MEM}$	Modified
N_j	Unconditional jump instructions	NA	Added
N_{br}^T	Taken branches	N_{br}^T	Same
d	Pipeline depth (branch penalty)	d	Same
N_{ld_deps}	Load-use hazard stall cycles	N_{ld_deps}	Same
N_{j_deps}	Jump-register hazard stall cycles	NA	Added
N_{ld}, N_{st}	Load / store instruction count	$N_{ld/st}$	Same
ℓ_{TCDM}	TCDM access latency (cycles)	$\ell_{Dcache\_hit\_time}$	Modified

The modified and newly added parameters reflect the absence of a cache hierarchy in Pulpissimo and the specific hazards present in the CV32E40P pipeline [33]:

- I_{miss} **(modified):** The original model computes instruction-miss penalty as an I-cache miss count multiplied by a fixed memory latency, $m_{Icache} \cdot \ell_{MEM}$, reflecting a cache-based hierarchy. With no instruction cache present, this is replaced by I_{miss}, a stall-cycle count measured directly by the CV32E40P's internal counters, representing cycles lost to TCDM bank contention during instruction fetch rather than a fixed-latency cache miss as shown in Figure 5.2.
- D_{stall} **(modified):** Analogously, the original data-miss penalty term, $m_{Icache} \cdot \ell_{MEM}$, is replaced by D_{stall}, measured directly via `fc_data_stall_cnt` (see Section 5.2).
- ℓ_{TCDM} **(modified):** With the absence of a cache hierarchy, the concept of a hit time doesn't exist. Instead, this is replaced by the minimum latency to access the TCDM bank data. ℓ_{TCDM} is fixed at 1 cycle, following directly from the TCDM handshake behaviour as shown in Figure 5.2.
- N_j **(added):** Accounts for unconditional jump instructions, which are architecturally distinct from conditional branches in the CV32E40P pipeline and were not separately represented in the original model. Jump instructions have a penalty of 1 cycle.
- N_{j_deps} **(added):** Accounts for stall cycles caused specifically by jump-register hazards, i.e. an indirect jump whose target register value depends on the result of an immediately preceding instruction, which the original model did

not require, as it did not separate jump instructions from branches.

- $N_{br}^{T} \cdot d$: There is no branch prediction in the CV32E40P core. Hence, all taken branches are mispredicted branches and suffer d cycles of penalty. d is fixed at 2 as per the pipeline details of the CV32E40P core [33].

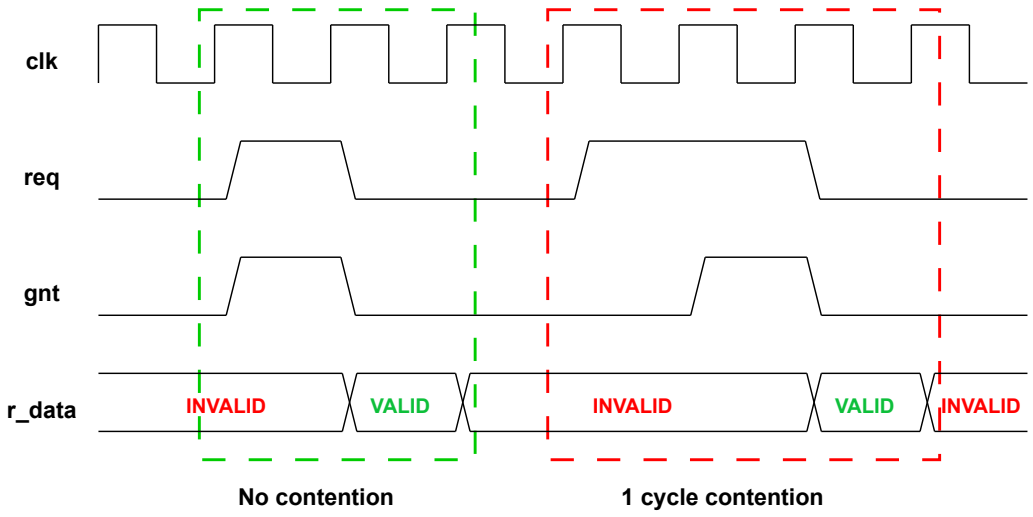

Figure 5.2: Timing diagram of TCDM read access showing non-contention and contention cases. In the no contention case, valid data is available 1 cycle after `req` assertion. Write access works in the same way.

5.1.3 Accelerator-Based System

The HWPE-MAC accelerator is a Hardware Processing Engine (HWPE) integrated into Pulpissimo's memory-mapped address space, sharing the TCDM interconnect with the core rather than communicating through a dedicated bus or DMA channel as seen in Figure 5.1. Architecturally, the accelerator is designed to compute the inner product of two one-dimensional vectors: it is configured with a base address and stride for each of the two input vectors and a length parameter, after which it autonomously streams the corresponding elements from TCDM through its multiply-accumulate datapath and returns a single scalar result. This vector-oriented design means the accelerator has no native notion of a matrix or of two-dimensional data reuse; it is only capable of producing one output value per invocation.

Consequently, computing a full $n \times n$ matrix multiplication requires the host to decompose the operation into n^2 independent row-column dot products, each corresponding to a single output element of the result matrix. Before each dot product, the host must reconfigure the accelerator by writing a new pair of base addresses – pointing to the appropriate row of the first matrix and column of the second – along with the vector length, to the accelerator's memory-mapped control registers. The accelerator is then triggered and executed to completion before the host issues the next job. This reconfigure-then-trigger sequence is repeated once per output element, so that offloading an $n \times n$ matrix multiplication results in n^2 separate accelerator invocations rather than a single bulk transfer of the full problem. To avoid

TCDM port contention between the core and the accelerator during this repeated reconfiguration, the two input matrices are placed in the interleaved TCDM banks with a deliberate address offset, ensuring that concurrent accesses from different HWPE ports do not collide on the same bank.

Adapting LogCA to the Accelerator's Dispatch Structure

Section 4.3.4 introduced the LogCA model in its standard form, with host and accelerator execution times given by

$$T_0(g) = C \cdot g^{\beta}, \qquad T_1(g) = o + L + \frac{C \cdot g^{\beta}}{A}, \tag{5.3}$$

where the overhead o is treated as a constant, independent of granularity. This assumption reflects systems in which offloading corresponds to a single, discrete act like a system call or a DMA transfer initiation, whose cost does not scale with the amount of data offloaded. The HWPE-MAC accelerator does not fit this assumption: because each output element requires a separate job dispatch, the total dispatch overhead scales with the number of jobs, and therefore with n^2. Since the granularity is defined as $g = 2n^2 \cdot \text{sizeof(element)}$ (the combined size of the two input matrices), the number of dispatches is directly proportional to g, and the constant-overhead assumption of the original model was replaced with a term linear in granularity:

$$T_1(g) = o_g \cdot g + L + \frac{C \cdot g^{\beta}}{A}, \tag{5.4}$$

where o_g denotes the dispatch overhead per byte of offloaded data (cycles/byte), rather than a one-time setup cost. This reformulation to a g-linear form preserves the granularity-based formulation of the original model while capturing the accelerator's actual dispatch behaviour.

The computational index C and complexity exponent β were derived following the same regression-based method described in Section 4.3.4: the matrix multiplication kernel was executed on the core alone across a range of matrix sizes, and C and β were obtained by fitting $\ln T_0(g) = \ln C + \beta \ln g$ to the measured execution times via least-squares regression. The interface latency L was not estimated empirically but fixed at its architectural minimum of one cycle assuming no bank contention. This assumption holds here by construction, given the staggered placement of the two input matrices described above and no other master accesses the TCDM ports during the streaming operation.

Because the dispatch overhead is now granularity-dependent rather than constant, it can no longer be isolated using the small-granularity limit employed for o in the original model (§4.2.5). The acceleration A was likewise not determined from the flattening of the speedup curve at large granularities, as prescribed by the original model. This was not a consequence of the overhead reformulation, but of a resource constraint on the target FPGA: the interleaved TCDM region that could be synthesised was limited to 256 KB, which bounded the largest matrix size – and therefore the largest achievable granularity – that could be evaluated. Within this

limit, the measured speedup did not reach saturation, so no plateau was observable from which A could be read directly. Instead, o_g and A were estimated jointly by minimising the sum of squared errors (SSE) between the modelled and measured $T_1(g)$ across all measured granularities,

$$\min_{o_g,\,A} \sum_i \left[T_1(g_i;\, o_g, A) - T_{1,\text{measured}}(g_i)\right]^2, \tag{5.5}$$

using the Levenberg-Marquardt iterative algorithm [34], with C, β, and L held fixed at their previously derived values.

Experimental Sweep Design

Following the sweep methodology used in the original LogCA evaluation, execution time was measured across a range of granularities by varying the matrix dimension n. Square matrices of size $n \in \{4, 8, 12, 16, 20, 28, 40, 56, 80, 112\}$ were used, corresponding to granularities ranging from 128 B to approximately 98 KB. The largest matrix size in the sweep was constrained by the capacity of the interleaved TCDM region, which was found to provide 256 KB (four banks of 64 KB each) of usable memory. Both the core-only and accelerator-offloaded implementations retain five resident $n \times n$ arrays of 4-byte elements in this region at any time (reference and working copies of the two input matrices, plus the output matrix), giving a total memory requirement of

$$5 \times 4 \times n^2 = 20n^2 \text{ bytes}, \tag{5.6}$$

which must satisfy $20n^2 \leq 262144$, or equivalently $n \leq \sqrt{262144/20} \approx 114.5$. The largest matrix size within this bound was fixed to $n = 112$, which was confirmed to occupy 250,884 B of the 262,144 B budget when inspecting the linked binary, leaving approximately 4.3% headroom.

5.2 Performance Monitoring on Pulpissimo

A dedicated peripheral, `perf_cnt_unit`, is added to observe TCDM activity by tapping the `req` and `gnt` signals of four interconnect ports: the core's data port (`fc_data`), the MAC accelerator's HWPE port (`hwpe-mac`), and the interleaved and private L2 memory banks (`l2_interleaved`, `l2_private`). The same peripheral is used for the accelerator-based system as well. Three counter types are implemented:

- `*_req_cnt` – increments whenever the port's `req` line is asserted, i.e. the number of requests issued.
- `*_stall_cnt` – increments whenever `req` is asserted but `gnt` is not, i.e. a request is outstanding due to contention.
- `*_util_cnt` – increments whenever `req` and `gnt` are both asserted, i.e. a successful access completes.

This gives `fc_data_req_cnt` and `fc_data_stall_cnt` for the core's data port, `hwpe_req_cnt` and `hwpe_stall_cnt` per HWPE port, and `l2_interleaved_util_cnt` / `l2_private_util_cnt` per memory bank. Figure 5.3 shows the tap points for these counters in the PULP architecture.

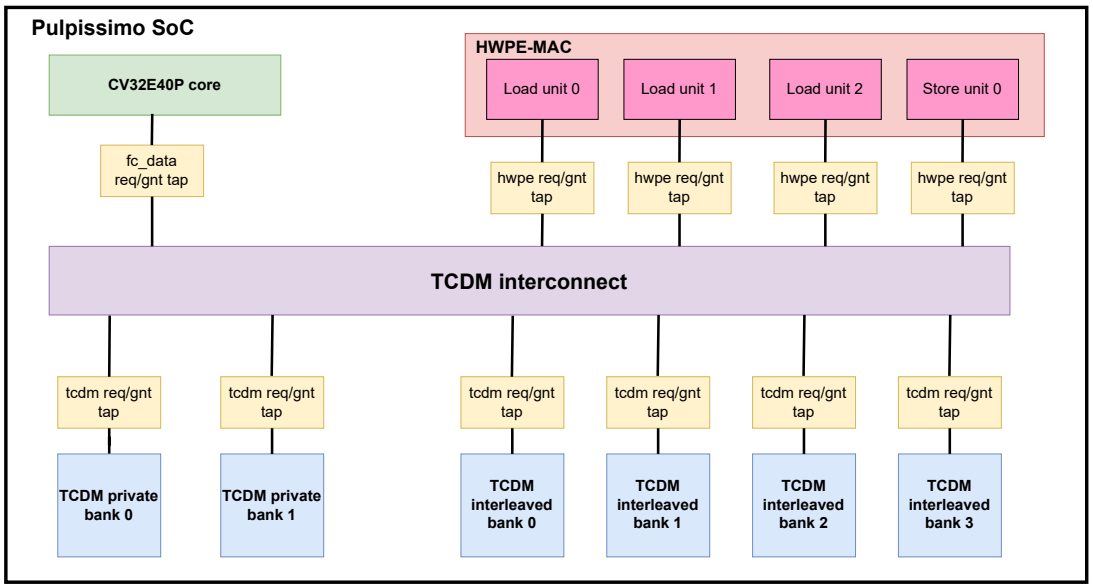

Figure 5.3: Tap points of proposed counters in the Pulpissimo SoC

In addition to these custom counters, the CV32E40P core used in this system provides internal counters for core architectural events. Mapping of model parameters to relevant counters is shown in Table 5.2. $T_{measured}$ is the total number of cycles measured by the core, used for assessing model accuracy.

The added counters are exposed as memory-mapped registers, reusing the existing APB peripheral bus interface: the APB address space allocated to existing peripherals is extended with a dedicated register range for `perf_cnt_unit` (dedicated RTL module for the added counter logic), requiring no new bus infrastructure. Counters are enabled just before the start of the benchmark and their values read directly by software running on the core itself on benchmark completion.

5.3 Benchmark Selection

Benchmark selection follows the reduced-workload rationale introduced in Section 4.1. The Embench benchmark suite [35] is used as the reference workload, as it was designed specifically for microcontroller-class embedded targets and reports a single aggregate score for cross-platform performance and code-size comparison, matching the class of system under evaluation in this case study.

Rather than executing the full suite, a single benchmark is selected from Embench for this case study. This choice reflects the goal of the work: the focus is on evaluating the performance monitoring *methodology* itself, rather than on characterising

Table 5.2: Mapping of performance model parameters to hardware counters

Parameter	Counter	Status
N	`minstret`	Existing (CV32E40P)
I_{miss}	`IMISS`	Existing (CV32E40P)
N_{br}^{T}	`BRANCH_TAKEN`	Existing (CV32E40P)
N_j	`JUMP`	Existing (CV32E40P)
N_{ld}	`LD`	Existing (CV32E40P)
N_{st}	`ST`	Existing (CV32E40P)
N_{ld_deps}	`LD_STALL`	Existing (CV32E40P)
N_{j_deps}	`JMP_STALL`	Existing (CV32E40P)
$T_{measured}$	`mcycle`	Existing (CV32E40P)
D_{stall}	`fc_data_stall_cnt`	Newly added (`perf_cnt_unit`)
—	`fc_data_req_cnt`	Newly added (`perf_cnt_unit`)
—	`hwpe_req_cnt` (per port)	Newly added (`perf_cnt_unit`)
—	`hwpe_stall_cnt` (per port)	Newly added (`perf_cnt_unit`)
—	`l2_interleaved_util_cnt` (per bank)	Newly added (`perf_cnt_unit`)
—	`l2_private_util_cnt` (per bank)	Newly added (`perf_cnt_unit`)

the SoC across a broad workload space. Different benchmarks stress different architectural parameters depending on whether they are compute- or memory-intensive, and evaluating multiple benchmarks would distribute effort across workload characterisation rather than methodology validation. A single, representative benchmark therefore keeps the case study focused while still exercising both systems under evaluation.

`matmult-int`, a 20×20 integer matrix multiplication, is selected as this benchmark. It is chosen for its simplicity, and because it maps directly onto the sole function of the MAC accelerator introduced in Section 5.1.3: the accelerator performs multiply-accumulate operations exclusively, and `matmult-int` consists almost entirely of the same operation, making it a natural fit for evaluating both the baseline core and the accelerator-augmented system with the same workload.

5.4 Experimental Environment

Carrying out the case study relies on a set of tools spanning software compilation, RTL dependency management, simulation, and FPGA implementation. Software for the target system is compiled using the PULP RISC-V GNU toolchain [36]. RTL dependencies across the many IP repositories that make up the Pulpissimo platform are resolved using Bender [37]. Functional verification, following the staged approach of Section 3.3, uses Mentor/Siemens QuestaSim for cycle-accurate RTL simulation,

and Xilinx Vivado for FPGA synthesis, implementation, and bitstream generation, targeting the KCU105 board as described in Section 5.4.2. Once a design is running on FPGA, OpenOCD provides the JTAG communication channel to the RISC-V core's debug module, and GDB is used on top of this connection to load programs into memory and debug their execution. UART prints were observed on host PC via minicom serial terminal.

5.4.1 RTL Simulation

Functional simulation is performed using QuestaSim, one of two RTL simulators supported out of the box by Pulpissimo (the other being Cadence Xcelium). Building and running a simulation follows a fixed sequence: resolve and download all required IPs via Bender, build the RTL platform to compile a self-contained simulation model of the platform, and a given test is then compiled and simulated with a make command from within its own test directory. The platform heavily relies on shell scripts to automate the processes and required editing due to errors when building the platform for the first time. The QuestaSim commands were also updated in the TCL script to enable signal dumping to verify the functionality of the proposed counters.

5.4.2 FPGA Implementation

Implementing Pulpissimo on a new FPGA target required adapting the existing open-source board support infrastructure rather than building a flow from scratch. The Pulpissimo platform provides board-specific configurations for a number of Xilinx targets, including the VCU108 and the Zedboard [1]; these were used as the starting point for bringing up the Xilinx KCU105. Retargeting the flow required updating the board's constraints file (`.xdc`) with pin assignments specific to the KCU105, along with corresponding changes to the Makefile and the Vivado TCL script responsible for project creation, so that a new KCU105 board configuration could be selected alongside the existing targets.

Bringing the design up on the new target surfaced a removal timing violation in the reset architecture. Nearly all blocks in the design were serviced by a single reset source, and at the destination flip-flops the clock signal arrived later than the (asynchronously de-asserted) reset signal relative to the active clock edge, violating the reset removal timing requirement. Correcting this by re-analysing and re-balancing the clock network to reduce destination-side clock slack was judged too complex and time-consuming for the scope of this work. Instead, a `MAX_FANOUT` constraint was applied to the reset signal, forcing the tool to replicate the source flip-flop and pipeline the reset distribution, resolving the violation without requiring manual restructuring of the clock tree.

After timing closure and implementation of the design on the KCU105, an issue was encountered when establishing a JTAG debug connection to the RISC-V core. These JTAG Test Access Port (TAP) signals are used to communicate with and debug the core via GDB, as introduced in Section 3.2. Existing board configurations expose the RISC-V JTAG TAP signals directly on GPIO pins for connection to an external

JTAG debugger; the same approach was applied to the KCU105 board configuration but failed to establish a connection via OpenOCD. Rather than debugging the external GPIO path directly, the design was switched to use a BSCANE2 primitive, which tunnels JTAG signals through the FPGA's own dedicated configuration scan chain instead of external pins. PULP already provides a SystemVerilog block implementing this BSCANE2-based tunnelling, so the dependency management tool (Bender) was reconfigured to compile the BSCANE2 variant of the JTAG interface in place of the direct-GPIO TAP used on the other boards, which resolved the connection issue.

5.5 Functional Verification

Functional verification of the baseline system, the custom performance counters and the MAC accelerator, follows the staged simulation-then-FPGA methodology described in Section 3.3: for each new RTL iteration, cycle-accurate simulation is used first to obtain maximum signal visibility and remove obvious bugs, and only once a block passes simulation is it carried forward to FPGA. Simulation results at each stage serve as the reference model against which the corresponding FPGA run is later checked.

Baseline Platform Verification

The existing Pulpissimo platform IP is not re-verified block by block in this thesis, since not all blocks provided by the platform are exercised by the case study. Instead, a simple "hello world" program is used to confirm the basic functionality required by every subsequent stage: the core, TCDM, interconnect, and clock/reset generation. This baseline version of the platform does not yet contain the counter architecture introduced in this thesis, and is used purely to confirm the unmodified system works correctly out of the box before any new logic is added.

Core-Internal Counter Verification

With the baseline confirmed, software was written to enable and read the CV32E40P's internal architectural counters, and run first in simulation, where the resulting counter values were checked manually against expectation. The test software performs a 1-D vector inner product on two vectors of length 16. The same software was then run on FPGA, and the results matched the simulation-verified expectation.

`perf_cnt_unit` Verification

The custom `perf_cnt_unit` peripheral (Section 5.2) was then added to the design and verified using the same staged approach: run first in simulation and checked against manually derived expected values, then run on FPGA, where the results matched both the simulation output and the manual expectation. As a further cross-check specific to the FPGA run, the core's `fc_data_req_cnt` counter was compared against `l2_private_util_cnt[0]` (the linker settings map data to bank

0 and instructions to bank 1) and found to match, confirming that the newly-introduced bank-utilization counters were consistent with the core's request counter for this simple workload.

Accelerator Integration

Finally, the MAC accelerator was integrated, and the test software was modified to offload the same vector inner product computation to the accelerator rather than computing it on the core. In simulation, this produced the expected result. On FPGA, however, the accelerator produced an incorrect output that did not match the simulation-verified expected value, indicating a problem specific to the accelerator's FPGA implementation rather than to the counter infrastructure or the surrounding platform, both of which had already been verified independently in the preceding stages.

Root-causing this discrepancy was supported by searching the Pulpissimo's public issue tracker [38], which identified a known issue: the accelerator's register-programming logic was implemented using latch-based storage, which does not function correctly when synthesised onto FPGA fabric, consistent with the ASIC-versus-FPGA primitive differences discussed in Section 3.1. The fix was to replace the latch-based implementation with an equivalent flip-flop-based one. Following this change, the FPGA output matched the simulation output for the same software, confirming the accelerator's functional correctness on FPGA.

Stall Counter Verification

The accelerator's per-port stall counters were verified using a negative test. Initially, the stall counters for the HWPE ports servicing the two source vector operands read a one-cycle stall, caused by the two vectors being placed concurrently in memory such that both operands mapped to the same TCDM bank in a given cycle. To resolve this, the two vectors were encapsulated in a structure with a padding integer inserted between them, breaking their concurrent placement and distributing the two operands across different banks. After this change, the stall counters correctly read zero, confirming that the counters accurately detect port contention when it is present and correctly report its absence once removed.

Through this staged process, functional coverage of the newly introduced counters was achieved: all counters were exercised and verified against either a simulation reference or a manually derived expected value, on both simulation and FPGA, and the stall counters were additionally confirmed to respond correctly to a deliberately introduced contention scenario and its subsequent removal.

6

Results

This chapter illustrates the results of the case study experiments done in Chapter 5.

6.1 Single-Core System

Table 6.1 reports the counter values obtained from the single-core baseline system executing `matmult-int` over 39 repetitions, together with each term's contribution to modelled execution time via equation 5.2, shown again as equation 6.1.

Table 6.1: Single-core counter breakdown (per repetition, `matmult-int`, 39 reps)

Parameter	Counter name	Per-rep value	% of modelled T
N	`MINSTRET`	37,147.9	**60.38%**
N_{br}^{T}	`BRANCH_TAKEN`	16,359.9	**26.59%**
N_{ld_deps}	`LD_STALL`	8,000.0	**13.00%**
N_{ld}	`LD`	16,800.1	—
N_{st}	`ST`	1,600.4	—
N_{j}	`JUMP`	19.1	0.03%
I_{miss}	`IMISS`	0	—
D_{stall}	`fc_data_stall`	0	—
N_{j_deps}	`JMP_STALL`	0	—

$$T = N + I_{miss} + D_{stall} + N_j + N_{br}^{T} \cdot d + N_{ld\_deps} + N_{j\_deps} + (N_{ld} + N_{st}) \cdot (\ell_{TCDM} - 1) \tag{6.1}$$

6.1.1 Bottleneck Discussion

- **Instruction count (N, 60.4%)** is set by the benchmark itself rather than by the architecture, and its dominance indicates that the bottleneck for this workload is not architecture-dependent in the current configuration. N could in principle be reduced by issuing single instruction, multiple data (SIMD) instructions on a more capable core, but this would require a fundamental change to both the ISA and the core architecture.
- **Taken branches (26.6%)** occur on every inner-loop iteration except the last one as seen from Figure 6.1 (PC:`0x1c0081c2`; `bne a4,a1,...`). The same

applies to the middle loop as well (PC:`0x1c0081cc`).This could be mitigated by introducing a branch predictor, since the branch is taken on the large majority of executions and is therefore highly predictable.

- **Load-use hazards (13.0%)** also occur on every iteration of the innermost loop. The innermost loop body (line 154 in Figure 6.1, the multiply-accumulate itself) compiles to a fixed three-instruction sequence: two loads (`p.lw t5,4(a1!)` and `p.lw t4,80(t1!)`) followed immediately by a multiply-accumulate (`p.mac a2,t5,t4`). Because `p.mac` consumes `t4` the cycle after it is loaded, and no independent instruction exists between the second load and the MAC to fill this gap, every iteration of the innermost loop incurs a load-use hazard stall. Software-level loop unrolling by using two iterations per inner-loop pass would allow the compiler to interleave independent load/MAC sequences and hide part of this stall, without requiring any architectural change.

6.1.2 Model Accuracy

Table 6.2 compares modelled and measured execution time.

Table 6.2: Model accuracy, single-core baseline

Quantity	**Value**
T_{actual} (measured, per rep)	69,128 cycles
T_{model} (Eq. 5.2, per rep)	61,527 cycles
Deviation	**−11.0%**

The model underestimates measured execution time by 11%, indicating that at least one architectural event contributing to T_{actual} is not yet captured by the current model. Confirming this requires reducing the benchmark's data size and cross-checking against a signal-level RTL simulation, where the source of the discrepancy can be observed directly rather than inferred from aggregate counter totals.

6.1.3 TCDM Port Utilization

Only the private TCDM banks are exercised by the single-core system; the interleaved banks record zero utilization, as no data was placed there for this configuration. By default, data accesses map to private bank 0 and instruction fetches map to private bank 1. Table 6.3 reports the total `util_cnt` values recorded over the full 39-repetition run.

Table 6.3: TCDM private bank utilisation, single-core baseline (39 reps total)

Bank	**Total util. count**	**% of total cycles**
Private bank 0 (data)	717,619	26.62%
Private bank 1 (instructions)	2,312,810	85.79%

The data-bank utilization count (717,619) matches the combined load-and-store count summed over all 39 repetitions almost exactly ($(N_{ld} + N_{st}) \times 39 \approx 717{,}620$),

confirming that every load and store instruction results in exactly one counted data-bank access, as expected from the counter design in §5.5.

The instruction-bank utilization count admits a similar check, but requires accounting for the branch penalty: each taken branch does not merely cost one redirected fetch, but $d = 2$ fetch-side cycles, consistent with the pipeline depth used elsewhere in the model (§5.5). The expected instruction-bank access count is therefore $N + 2 \cdot N_{br}^{T} + N_j$, rather than $N + N_{br}^{T} + N_j$. Evaluated over the full 39-repetition run, this gives:

$$N + 2 \cdot N_{br}^{T} + N_j = 1{,}448{,}769 + 2(319{,}019) + 746 = 2{,}087{,}553 \tag{6.2}$$

against a measured instruction-bank utilization of 2,312,810 – the formula falls short of the measured count by approximately 9.7%. This closely matches the 11% deviation between modelled and measured execution time reported above (Table 6.2), suggesting that the two discrepancies share a common cause: additional fetch-side cycles associated with taken branches that are not yet fully accounted for in either the execution-time model or this reconstruction of instruction-bank traffic.

6.1.4 Resource Cost

Table 6.4 reports the resource cost of the counter infrastructure added for the single-core baseline.

Table 6.4: Area cost, single-core baseline

Component	**Logic LUTs**	**FFs**	**DSP**
Total used	49,427	27,661	16
`perf_cnt_unit`	498	513	0
% of total design	**1.01%**	**1.85%**	**0.00%**

The instrumentation overhead of the baseline counter set is small relative to total design area, at under 2% of both LUTs and FFs and no DSP usage. The significance of this cost relative to the diagnostic value obtained is discussed further in Section 7.4.

```
144   void
145   Multiply (matrix A, matrix B, matrix Res)
146   {
147     register int Outer, Inner, Index;
148
149     for (Outer = 0; Outer < UPPERLIMIT; Outer++)
        0x1c0081a0 <+154>:  mv    a7,t0
        0x1c0081a2 <+156>:  mv    a6,s3

150       for (Inner = 0; Inner < UPPERLIMIT; Inner++)
        0x1c0081a4 <+158>:  mv    a5,t6
        0x1c0081a6 <+160>:  mv    a3,a6
        0x1c0081a8 <+162>:  addi  a4,a7,80

151       {
152         Res[Outer][Inner] = ZERO;
        0x1c0081ac <+166>:  p.sw  zero,4(a3!)

153         for (Index = 0; Index < UPPERLIMIT; Index++)
        0x1c0081b0 <+170>:  mv    t1,a5
        0x1c0081b2 <+172>:  mv    a1,a7
        0x1c0081b4 <+174>:  li    a2,0
154           Res[Outer][Inner] += A[Outer][Index] * B[Index][Inner];
        0x1c0081b6 <+176>:  p.lw  t5,4(a1!)      <-- load
        0x1c0081ba <+180>:  p.lw  t4,80(t1!)     <-- load
        0x1c0081be <+184>:  p.mac a2,t5,t4       <-- load-use hazard!

153         for (Index = 0; Index < UPPERLIMIT; Index++)
        0x1c0081c2 <+188>:  bne   a4,a1,0x1c0081b6 <benchmark_body+176>
        0x1c0081c6 <+192>:  sw    a2,-4(a3)

150       for (Inner = 0; Inner < UPPERLIMIT; Inner++)
        0x1c0081ca <+196>:  addi  a5,a5,4
        0x1c0081cc <+198>:  bne   a5,t2,0x1c0081ac <benchmark_body+166>

149     for (Outer = 0; Outer < UPPERLIMIT; Outer++)
        0x1c0081d0 <+202>:  addi  a6,a6,80
        0x1c0081d4 <+206>:  beq   a4,s0,0x1c0081dc <benchmark_body+214>
        0x1c0081d8 <+210>:  mv    a7,a4
        0x1c0081da <+212>:  j     0x1c0081a4 <benchmark_body+158>
```

Figure 6.1: Disassembled inner loop of `matmult-int`, showing the load-use hazard between the two `p.lw` instructions and the dependent `p.mac`, and the taken branch (`bne`) closing each loop level.

6.2 Accelerator-Based System

Since offloading a compute-intensive kernel to a dedicated accelerator is a well-established way to obtain speedup, observing a speedup here is not in itself a novel

result. The emphasis of this section is instead on the evaluation strategy used to characterise *how* that speedup is achieved and where it is limited – fitting the LogCA model to measured data, identifying the resulting bottlenecks, and deriving concrete optimisation directions from the fitted parameters, rather than on the existence of speedup alone.

Table 6.5 lists the LogCA parameters fitted for the MAC-accelerator system, obtained via the procedure described in Section 5.1.3.

Table 6.5: Fitted LogCA parameters, MAC accelerator

Parameter	**Symbol**	**Value**
Overhead (per-byte dispatch cost)	o_g	5.255 cycles/byte
Computational index	C	0.102 cycles/byte
Acceleration	A	9.46
Latency	L	1 cycle
Scaling exponent	β	1.609

6.2.1 Speedup Curve: Model vs. Measured

Figure 6.2 shows measured speedup $T_0(g)/T_1(g)$ against the fitted g-linear model, extended well beyond the measured matrix size range ($n = 4$ to $n = 112$) toward saturation, with the fitted parameters from Table 6.5. This reveals that saturation starts occurring at 128 MB data size, which corresponds to a matrix size of 4096.

The break-even granularity, $g_1 \approx 800$ B, is the point at which speedup crosses 1: below g_1, the accelerator provides no benefit over the single-core system, since the fixed accelerator configuration overhead $o_g \cdot g$ has not yet been amortised by the accelerator's throughput advantage. The half-acceleration granularity, $g_{A/2} \approx 26$ KB, is the point at which half of the peak acceleration ($A/2 \approx 4.731$) is reached; notably, this occurs at only around 25% of the largest granularity actually tested ($g = 100{,}352$ B, $n = 112$), meaning that for the majority of the offload sizes evaluated in this case study, the achievable speedup remains well below the accelerator's theoretical maximum. This has a direct practical implication: unless the offloaded data volume comfortably exceeds $g_{A/2}$, the accelerator delivers only a fraction of its peak acceleration, regardless of how favourable A itself is.

6.2.2 Model Accuracy

Table 6.6 reports the per-point prediction error of the fitted model against measured $T_1(g)$.

The largest deviation occurs at the smallest measured granularity ($n = 4$, $g = 128$ B), where the model underestimates T_1 by 32.4%. Prediction error shrinks monotonically as granularity increases, crossing zero between $n = 28$ and $n = 40$, and converging to under 0.5% for $n \geq 80$, with a near-exact match at the largest measured point ($n = 112$, +0.02%). The model therefore systematically

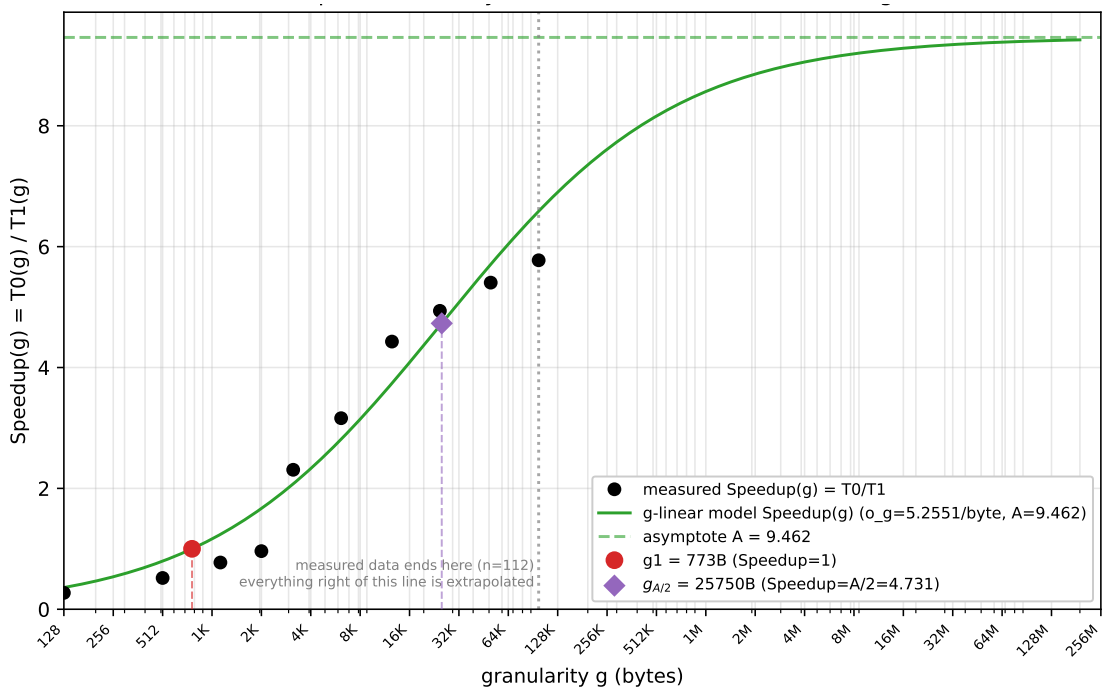

Figure 6.2: Measured vs modelled speedup curve. The modelled curve is extended beyond the measured granularities until 95% of theoretical saturation

underestimates execution time at small granularity and is most accurate at large granularity.

A holdout validation, refitting the model on 8 of the 10 points and predicting the two held-out points ($n = 20$ and $n = 56$), is reported in Table 6.7.

Holdout error at $n = 56$ (large granularity) closely matches its in-sample counterpart (+1.76% vs. +1.30%), indicating the fit generalises well in this region. At $n = 20$ (small-to-mid granularity), holdout error is similar (−18.85% vs. −19.72% at $n = 20$ in-sample), suggesting the model's reduced accuracy at small granularity is a structural limitation of the g-linear form itself, rather than overfitting to the specific 10 measured points.

6.2.3 Bottleneck Analysis

As per [32], a parameter is classified as a design bottleneck if a 10× improvement in it yields at least a 20% improvement in speedup, over some region of the granularity sweep. L is excluded from this test: it is already pinned at the physical floor of 1 cycle (the TCDM's documented single-cycle access latency under no contention), so a "10× improvement" (0.1 cycles) is not physically meaningful. Figure 6.3 extends the fitted model beyond the measured range toward speedup saturation, comparing baseline and 10×-improved speedup for each parameter individually.

At small granularity, C and o_g dominate: overhead-related terms are the limiting factor when little data is offloaded per call. At large granularity, all three parameters remain bottlenecks, with A becoming the largest single contributor; the accelerator's

Table 6.6: g-linear model validation: measured vs. predicted $T_1(g)$, all 10 points

n	g (bytes)	Measured T_1	Predicted T_1	% error
4	128	1,036	700.2	−32.41%
8	512	4,070	2,939.0	−27.79%
12	1,152	9,115	6,967.0	−23.57%
16	2,048	16,193	13,065.7	−19.31%
20	3,200	26,829	21,538.4	−19.72%
28	6,272	52,251	46,903.2	−10.24%
40	12,800	106,437	111,206.5	+4.48%
56	25,088	258,237	261,604.5	+1.30%
80	51,200	680,397	678,022.1	−0.35%
112	100,352	1,734,735	1,735,094.1	+0.02%

Table 6.7: Holdout validation: predicted (from 8-point fit) vs. measured, for held-out $n = [20, 56]$

n	g (bytes)	Measured T_1	Predicted T_1	% error
20	3,200	26,829	21,772.1	−18.85%
56	25,088	258,237	262,781.2	+1.76%

peak throughput becomes the limiting factor once the g-linear overheads are amortised. The overhead term does not become negligible even at the largest measured granularity: because $o_g \cdot g$ grows linearly with g while the compute term grows as $g^{1.609}$, the compute term is only around 2.3× the overhead term at $g = 100{,}352$ B – not yet large enough to make o_g irrelevant, consistent with it still appearing as a bottleneck at $n = 112$ in Figure 6.3. This also suggests that the rate of speedup increase might be slower compared to an overhead term independent of g.

Optimisation Suggestions

- **Reduce o_g.** Convert overhead from a per-byte to a constant cost by extending the accelerator's control interface to accept additional parameters per call, eliminating the linear scaling of overhead with granularity.
- **Increase A.** Add one additional compute unit; the current bank count supports exactly one more unit before TCDM port contention becomes limiting.
- **Improve C.** C is more naturally interpreted through computational intensity C/L, i.e. work done by the host per byte of offloaded data before dispatch; a higher C implies the host does more work per offload call while o_g and L remain fixed. This is of limited value for a standalone matrix multiplication, but would matter if the result were further processed on-accelerator (e.g. fused with a subsequent kernel) rather than returned to the host for computation.

6.2.4 TCDM Utilization

Figure 6.4 reports average interleaved TCDM bank utilization – accesses per bank per cycle, averaged across all interleaved banks – as a function of granularity, for

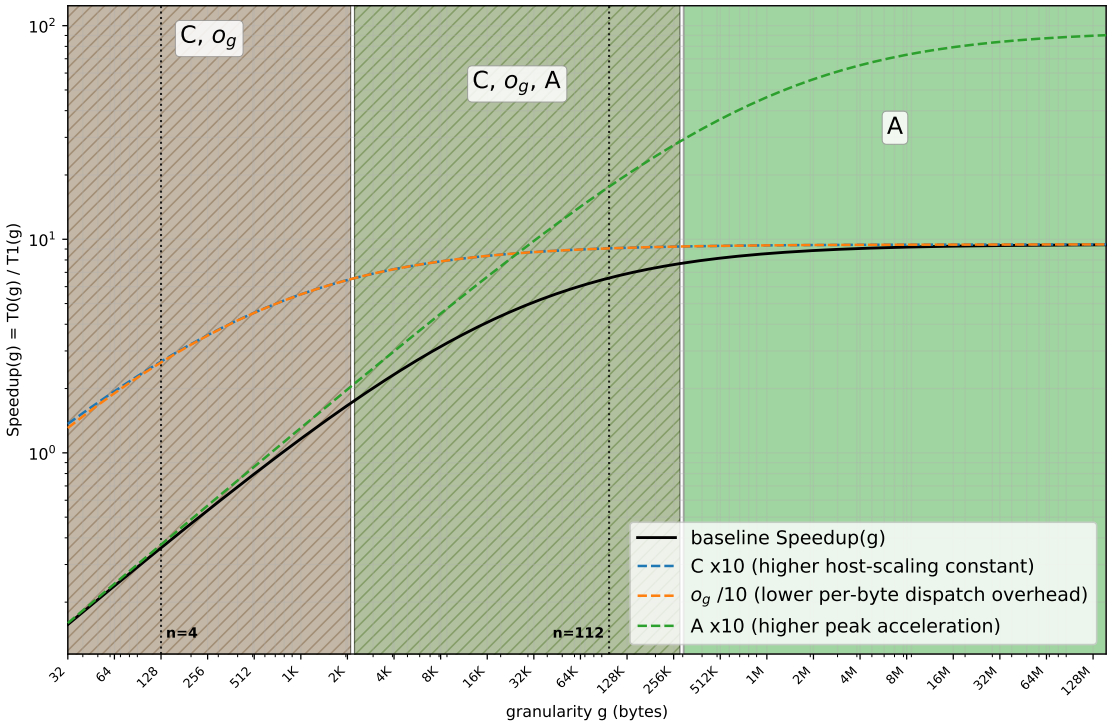

Figure 6.3: LogCA bottleneck identification: baseline vs. 10x-improved Speedup(g), all parameters combined. The blue curve closely follows the orange curve.

the host-only and accelerator-augmented systems. Figure 6.5 shows the derivative of this curve with respect to g, making the rate-of-change behaviour easier to read than from the raw curve alone.

Host-Only System

The host-only system's bank utilisation is largely insensitive to granularity once g is large enough for the benchmark's steady-state loop to dominate total execution time: utilisation converges to a flat plateau of approximately 0.073 accesses/bank/cycle for $g \gtrsim 3.2$ KB, with the derivative in Figure 6.5 correspondingly flattening to near zero over the same range. This is consistent with the disassembly code in Figure 6.1: once the fixed load/load/MAC/store instruction sequence dominates, the fraction of cycles involving an interleaved-bank access is an intrinsic property of that instruction mix, independent of how many times the loop executes.

At small granularity ($g = 128$–512 B), utilization instead dips before rising toward the plateau, and the derivative shows a corresponding sharp negative-to-positive swing. This suggests a transient behaviour when fixed-cost benchmark setup and initialisation code is small enough to contribute a non-negligible fraction of total cycles.

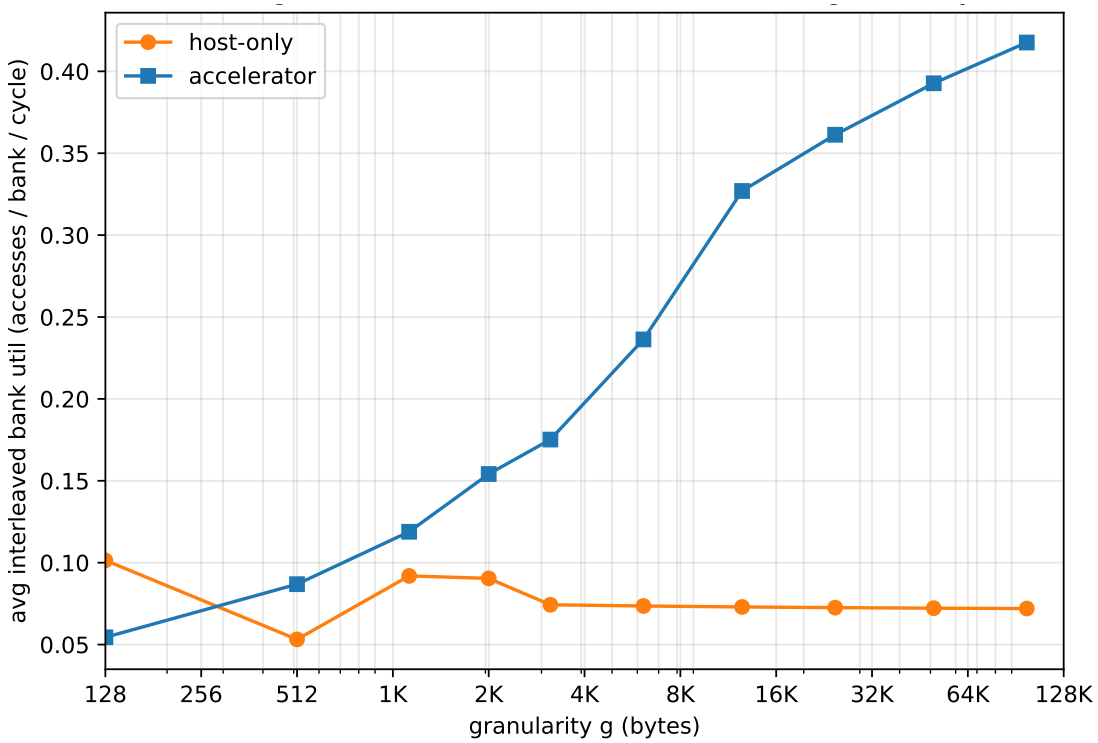

Figure 6.4: Average interleaved TCDM bank utilization vs. granularity, host-only vs. accelerator

Accelerator-Augmented System

In contrast to the host-only system, the accelerator's bank utilisation rises monotonically across the entire measured range, from approximately 0.055 at $g = 128$ B to 0.417 at $g = 100{,}352$ B, without reaching a plateau. The derivative in Figure 6.5 is positive throughout but consistently decreasing, indicating diminishing – but not yet exhausted – marginal growth in utilization as g increases.

This behaviour is consistent with the overhead/compute decomposition established in Section 6.2.3: total accelerator execution time is split between a dispatch/overhead phase ($o_g \cdot g$, largely control-interface traffic rather than TCDM data traffic) and a compute phase ($(C/A) \cdot g^{\beta}$, during which operands and results stream through the interleaved banks). As g grows, the compute phase's share of total cycles increases, since $\beta > 1$ causes it to grow faster than the overhead phase; a larger share of total cycles therefore falls within the TCDM-active compute window, driving average bank utilisation upward.

6.2.5 Resource Cost

Table 6.8 reports the resource cost of the MAC accelerator with respect to total resources used.

Unlike the counter-only overhead in Section 6.1.4, the accelerator itself is a substantial fraction of total design resources – over a quarter of available DSP resources

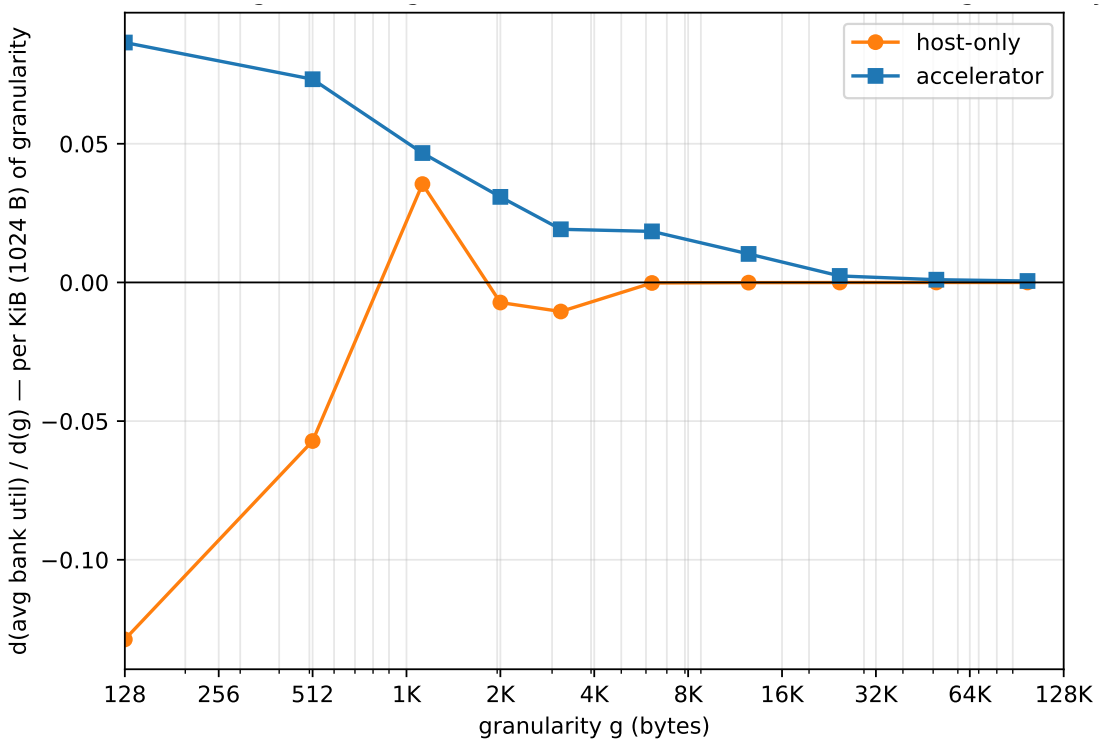

Figure 6.5: Rate of change of average interleaved TCDM bank utilisation vs. granularity, host-only vs. accelerator. Derivative computed via central differences (non-uniform spacing), except at the two boundary points, which use a one-sided estimate.

Table 6.8: Area cost, MAC-accelerator system

Component	**Logic LUTs**	**FFs**	**DSP**
Total used	50,300	29,661	16
`fc_hwpe`	5,681	5,290	4
% of total design	**11.29%**	**17.83%**	**25.00%**

and roughly a sixth of FFs – which should be weighed against the measured 9.46× peak acceleration and the granularities at which that acceleration is actually realised. This accelerator-based design with the proposed counter infrastructure still consumes only a fraction of the total device resources available, as seen in Table 6.9. The significance of this cost relative to the diagnostic and performance value obtained is discussed further in Section 7.4.

Table 6.9: Full device resource utilisation, MAC-accelerator system (Xilinx KCU105)

Resource	**Utilization**	**Available**	**Utilization %**
LUT	50,312	242,400	20.76%
FF	29,661	484,800	6.12%
BRAM	82	600	13.67%
DSP	16	1,920	0.83%

7

Discussion

7.1 Performance Bottlenecks

The case study's two systems illustrate how the focus of a performance model shifts depending on the device under test (DUT), and how a single evaluation methodology accommodates that shift without needing to be redesigned for each new system.

For the single-core baseline, the model decomposed execution time into core-internal architectural events – retired instructions, taken branches, and load-use hazards (Eq. 5.2). Interconnect latency was not a candidate bottleneck in this decomposition: the TCDM's fixed single-cycle access latency, absence of contention under this workload, and the core's single-issue in-order pipeline together meant that memory-side effects contributed negligibly to modelled execution time. The bottlenecks identified (instruction count, branch penalty, and load-use hazards) were therefore all internal to the core, and the optimisation suggestions that followed (branch prediction, loop unrolling) were correspondingly core-scoped.

Moving to the accelerator-augmented system, the bottleneck-relevant parameters shift from core-internal events to SoC-level integration parameters: dispatch overhead (o_g), interconnect latency (L), and peak acceleration (A). This shift is not specific to the particular accelerator chosen. Instead, it reflects a structural change in where execution cycles are spent once a component is offloaded from the core to a separate hardware block. The relevant bottleneck moves from the internals of the executing unit to the interface between it and the rest of the system.

In this case study, once o_g and A were identified as bottlenecks (Section 6.2.3), they were not broken down further. The accelerator's internal datapath was treated as a single unit for the purposes of this analysis. For more complex systems, this need not be the end point. Once a parameter such as A or o_g is identified as a bottleneck at the system level, it can itself be decomposed into the performance model of the underlying component. This follows the same top-down approach used to move from the abstract to fine-grained models, as seen in Section 4.4. For example, consider a NoC interconnect identified as a system-level bottleneck, rather than a point-to-point link. A NoC-specific model, covering router latency, buffer occupancy, and injection rate, could then be applied to identify which NoC-internal parameter is responsible for the bottleneck. This mirrors the same core-internal decomposition applied to the single-core system in the case study. This staged, recursive applicability is what makes the proposed methodology portable and scalable across DUTs

of increasing complexity, rather than requiring a bespoke model to be derived for every system from the outset.

7.2 Performance Monitoring on FPGA

The performance monitoring approach used in this case study is *static*: counters are read at the end of a fixed benchmark run, producing an aggregate measurement used offline to fit and validate a performance model. This mirrors how the methodology has been applied throughout this thesis – as a way to characterise system behaviour on real hardware rather than in simulation, replacing the abstraction of software timing models with directly measured, cycle-accurate counter data.

Static monitoring of this kind is well suited to design-time characterisation and model validation, but it is not the only use case for on-chip performance counters. A complementary application is *dynamic* (runtime) monitoring, in which performance models – of the same kind derived and validated here – are evaluated continuously during execution and used to drive runtime decisions rather than only informing offline analysis. Pienaar et al.'s Model-Driven Runtime (MDR), a runtime framework [39], demonstrates this for heterogeneous platforms that uses performance models to make key scheduling decisions during execution: which processing element a task should be mapped to, how tasks should be scheduled on a given processing element, and when data should be copied between memory spaces. This class of application was not explored in this case study, which focused on offline model validation rather than runtime decision-making. The same counter infrastructure developed here (the memory-mapped `perf_cnt_unit` and TCDM-tap counters) could in principle be read continuously to support this kind of runtime use case without requiring new instrumentation to be added.

A further point of significance is that dynamic monitoring of this kind is not FPGA-specific: a lightweight, always-on counter-based monitoring system of the kind used here is synthesisable and low-overhead enough to be carried forward into an ASIC implementation of the same design, where it could continue to support runtime model-driven decisions post-tape-out.

7.3 Open-Source SoCs

The most direct advantage to using an open-source SoC platform is free access to, and the ability to modify, the underlying system IP. This is what made the instrumentation approach used throughout this thesis possible in the first place: custom performance counters could be integrated directly into the RTL of an existing SoC, something not generally achievable with closed, proprietary IP. Pulpissimo is released under the Solderpad Hardware License v0.51, a permissive license closely modelled on Apache 2.0, which explicitly permits commercial use, modification, and redistribution without a copyleft obligation to open-source derivative modifications. This is directly relevant to publishing the work carried out here: the counter and control-interface modifications made to the platform in this thesis can be freely

described, distributed, or built upon without licensing conflict.

A second advantage is access to an open community around the platform. When debugging platform-level issues, as encountered during the KCU105 bring-up, being able to search and reference community-reported issues on the platform's public issue tracker can meaningfully narrow down root causes that would otherwise require time-consuming debugging.

Together, these advantages are particularly significant for academia and small and medium-sized enterprises (SMEs), where resource constraints often rule out large-scale simulation infrastructure or proprietary IP licenses that a well-resourced commercial design team could otherwise draw on. An open-source platform combined with FPGA-based prototyping, as used throughout this thesis, substantially lowers the barrier to meaningful architectural evaluation for such groups.

These advantages come with a corresponding lack of the supporting ecosystem typically available around mature, commercially-maintained platforms. Documentation for open-source SoC platforms can be out of date relative to the current state of the RTL. For example, the memory map details about the Pulpissimo SoC available on its GitHub page [1] were outdated and needed RTL inspection before corresponding software could be implemented. Tooling is a further source of friction: open-source platforms typically rely on open-source simulation and debug tools (e.g. Verilator, GDB, OpenOCD) that may lack native or well-tested support for a given platform's specific configuration, as encountered with the OpenOCD/JTAG connectivity issue on the KCU105, where the existing direct-GPIO debug approach did not work out of the box and required switching to a BSCANE2-based tunnelling scheme instead.

7.4 Performance Counter Overheads

The area cost of instrumentation, reported in Sections 6.1.4 and 6.2.5 alongside each system's functional area, has a direct bearing on FPGA selection for a prototyping effort of this kind. The area of the base design itself, as a percentage of the total resources available on a candidate device, sets a lower bound: an FPGA that cannot fit the design under evaluation is unusable regardless of instrumentation. In this case study, that base design comfortably fits on the target device, leaving headroom for the performance monitoring counters and the MAC accelerator to be added on top.

This margin cannot be assumed for every project. For larger base designs, the combined area of the design and its performance monitoring counters, rather than the design alone, is the number that should guide FPGA selection. This case study illustrates why the combined figure matters: the counter infrastructure added for the single-core baseline cost under 2% of LUTs and FFs (table 6.4), but the MAC accelerator and its associated counters together consumed over a quarter of available DSP resources and close to a fifth of FFs (table 6.8). A design with less initial headroom than the one used here could easily be pushed past device capacity once instrumentation is included, if only the base design's area is considered during device selection.

The specific composition of a device's resources, not only its total capacity, is also

relevant to selection. A compute-heavy application, such as the MAC accelerator, places disproportionate demand on DSP slices, whereas a memory-heavy application would instead be constrained primarily by available BRAM. Selecting a device with an unfavourable balance of resource types for the application under test can leave one resource type exhausted while others sit underused, even where total logic area would otherwise be sufficient.

Working Within Limited FPGA Resources

Small development teams, with access to only a single, fixed FPGA device, may not have the option of selecting a larger part when the combined design and counter area exceeds what is available. One practical solution is to partition the counter set itself across multiple synthesis runs: different subsets of counters are implemented and run in separate executions of the same benchmark, and the resulting values are combined afterward into a single overall picture of system behaviour. This trades measurement convenience for device compatibility, but comes with real limitations. Debugging becomes more difficult when counter values that would ideally be examined together, for example to correlate a stall on one port with a request on another in the same cycle, are instead captured in separate runs and only reconciled after the fact. It is also time-consuming: a test that would otherwise require a single execution must instead be repeated once per counter subset, multiplying both simulation and FPGA runtime for what is conceptually one measurement.

A more involved alternative is to partition the design itself across multiple FPGAs, splitting the system such that no single device needs to hold the full design plus its complete counter set. This avoids the repeated-execution overhead of the counter-subsetting approach, since all counters can in principle be active in a single run, but introduces its own complexity: inter-FPGA communication must be designed and verified, and the timing behaviour of a partitioned system can differ from that of the same design implemented on a single device, potentially affecting the very measurements being collected.

Finally, both accessibility and observability are worth weighing directly against area cost when deciding how much instrumentation to include. A larger counter set gives finer-grained observability into system behaviour, as demonstrated throughout Chapter 6, but each additional counter is also additional area that must be accessible to a design team within their available FPGA resources. As seen in Section 5.2, the counters were accessed via memory-mapped registers, which required extending the APB interface. In cases where APB extension is not possible due to resource constraints, other accessibility methods such as JTAG or debug module-based access can be used.

8

Conclusion

This thesis proposed and evaluated a methodology for functional verification and performance monitoring of open-source SoCs using FPGA-based PoC prototyping, combining a staged simulation-then-FPGA verification flow with a top-down performance modelling approach in which fine-grained instrumentation is added only where an abstract model identifies a bottleneck. The methodology was applied to Pulpissimo, ported to a new FPGA target, across two systems: a single-core baseline and the same system augmented with a MAC accelerator. For the single-core baseline, a literature model was extended with terms not present in its original form and mapped onto a combination of existing and proposed hardware counters, reproducing measured execution time closely enough to attribute the dominant sources of execution time with confidence, while leaving a modest deviation only partially explained. For the accelerator-augmented system, the LogCA model was adapted to account for behaviour specific to this platform, namely overhead that scales with offload granularity and parameter estimation under a memory-constrained offload range, and used to identify how the limiting factor on achievable speedup shifts with granularity. Across both systems, the bottleneck focus correctly shifted from core-internal to SoC-integration effects, as intended by the methodology's staged design, demonstrating that the approach is workable in practice. These results should, however, be read against scope limitations: scalability was demonstrated over a single scale-up step rather than at larger scale; the case study used one benchmark on a small, resource-constrained configuration; and the accelerator's measurable offload range was constrained by on-chip memory capacity well short of the point at which the fitted model predicts speedup saturation. Addressing these constraints motivates the priorities for future work. Applying the same methodology to a different open-source SoC platform would also test whether the counter-integration approach and model adaptations generalise beyond the specific architecture used in this case study. Beyond this case study, the same counter infrastructure could support dynamic, runtime-driven use of the performance models developed here rather than only offline analysis, and the methodology itself could be extended to counter sets exceeding single-FPGA capacity through subsetting or partitioning, or evaluated against alternative counter-access mechanisms to the memory-mapped approach used in this thesis.

Bibliography

[1] P. platform. (2024) Pulpissimo. Accessed: Apr. 9, 2026. [Online]. Available: https://github.com/pulp-platform/pulpissimo

[2] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini, "Pulp: A parallel ultra low power platform for next generation iot applications," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, 2015, pp. 1–39.

[3] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, "Openpiton: An open source manycore research framework," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 217–232. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872414

[4] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[5] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.

[6] ARM Ltd., "Arm coresight base system architecture," ARM, Cambridge, UK, Technical Report, 2022, version 5.2. [Online]. Available: https://developer.arm.com/documentation/den0068/v/?lang=en

[7] W. Snyder, "Verilator: Fast, Free, But for Me?" https://veripool.org/papers/Verilator_Fast_Free_Me_DVClub10_pres.pdf, sep 2010, presented at DV Club (DVClub10). Available as a PDF.

[8] P. D. Schiavone, E. Sanchez, A. Ruospo, F. Minervini, F. Zaruba, G. Haugou, and L. Benini, "An open-source verification framework for open-source cores: A risc-v case study," in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2018, pp. 43–48.

[9] K. Qin, X. Guo, M. Schulz, and C. Trinitis, "Feriver: An fpga-assisted emulated framework for rtl verification of risc-v processors," in *Proceedings of the 22nd ACM International Conference on Computing Frontiers*, ser. CF '25.

New York, NY, USA: Association for Computing Machinery, 2025, p. 132–140. [Online]. Available: https://doi.org/10.1145/3719276.3725174

[10] T. Szymkowiak, "FPGA-Based Prototyping of a Modern MPSoC," Master's Programme in Information Technology, Tampere University, Tampere, Finland, 2024, accessed via the direct PDF link: https://trepo.tuni.fi/bitstream/handle/10024/155184/SzymkowiakThomas.pdf?sequence=2&isAllowed=y. [Online]. Available: https://trepo.tuni.fi/handle/10024/155184

[11] Y. Kim, H.-S. Kim, R. Lee, and S. Kang, "FPGA-based verification methodology of SoC-type CMOS image signal processor," in *2009 IEEE International SOC Conference (SOCC)*, 2009, pp. 231–234.

[12] M. Jirsak, H. Siemen, J. Lienke, M. Grabmann, E. Schäfer, and G. Gläser, "Under cover: On-fpga coverage monitoring by netlist instrumentation," in *2023 19th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, 2023, pp. 1–4.

[13] P. Larsson-Edefors and E. Börjeson, "Fiber-on-Chip: Digital FPGA Emulation of Channel Impairments for Real-Time Evaluation of DSP," in *2022 Optical Fiber Communications Conference and Exhibition (OFC)*, 2022, pp. 1–3.

[14] A. G. Schmidt, N. Steiner, M. French, and R. Sass, "HwPMI: An Extensible Performance Monitoring Infrastructure for Improving Hardware Design and Productivity on FPGAs," *International Journal of Reconfigurable Computing*, vol. 2012, pp. 162 404:1–162 404:15, 2012. [Online]. Available: https://doi.org/10.1155/2012/162404

[15] M. S. Jafri, "A Centralized System Performance Monitoring Infrastructure," Master's Thesis, University of Waterloo, Waterloo, Ontario, Canada, 2024, accessed via the direct PDF link: https://dspacemainprd01.lib.uwaterloo.ca/server/api/core/bitstreams/32f97622-ba64-4c3b-87d8-c669c2de6012/content. [Online]. Available: https://uwspace.uwaterloo.ca/handle/10012/8660

[16] A. Mirsalari. (2025) Pulp. Accessed: Apr. 6, 2026. [Online]. Available: https://github.com/ahmad-mirsalari/PULP

[17] D. Amos, A. Lesea, and R. Richter, *FPGA-Based Prototyping Methodology Manual: Best Practices in Design-For-Prototyping*. Mountain View, CA: Synopsys Press, 2011.

[18] L. Eeckhout, *Computer Architecture Performance Evaluation Methods*. Springer International Publishing, 2010. [Online]. Available: https://doi.org/10.1007/978-3-031-01727-8

[19] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Cambridge, MA: Morgan Kaufmann, 2019.

[20] M. Parelkar, *Demystifying the Digital Design Interview: The Missing Guide for Practical RTL and FPGA Interview Preparation*. Notion Press, 2026.

[21] Tektronix, "FPGA debug fundamentals," Application note/primer. [Online]. Available: https://www.tek.com/en/documents/primer/fpga-debug-fundamentals

[22] AMD/Xilinx, *Vivado Design Suite User Guide: Programming and Debugging (UG908)*, AMD, 2024, v2024.2.

[23] T. Newsome and M. Wachs, "RISC-V external debug support," RISC-V Foundation, Tech. Rep., 2024, version 0.13.2.

[24] OpenOCD Project, *Open On-Chip Debugger: OpenOCD User's Guide*, 2024, release 0.12.0.

[25] EnjoyDigital, "LiteScope: Small footprint and configurable embedded FPGA logic analyzer," GitHub repository. [Online]. Available: https://github.com/enjoy-digital/litescope

[26] J. J. Yi, R. Sendag, L. Eeckhout, A. Joshi, D. J. Lilja, and L. K. John, "Evaluating benchmark subsetting approaches," in *Proceedings of the 2006 IEEE International Symposium on Workload Characterization (IISWC)*, 2006, pp. 93–104.

[27] M. Breughe, Z. Li, Y. Chen, S. Eyerman, O. Temam, C. Wu, and L. Eeckhout, "How sensitive is processor customization to the workload's input datasets?" in *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*, 2011, pp. 1–7.

[28] T. Karkhanis and J. Smith, "A first-order superscalar processor model," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, 2004, pp. 338–349.

[29] M. Breughe, S. Eyerman, and L. Eeckhout, "A mechanistic performance model for superscalar in-order processors," in *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, 2012, pp. 14–24.

[30] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.

[31] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, p. 65–76, Apr. 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785

[32] M. S. B. Altaf and D. A. Wood, "Logca: A performance model for hardware accelerators," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 132–135, 2015.

[33] OpenHW Group, *CV32E40P User Manual*, OpenHW Group, 2024, version 1.8.3. Accessed: 2026-08-17. [Online]. Available: https://docs.openhwgroup.org/projects/cv32e40p-user-manual/en/latest/index.html

[34] J. J. Moré, "The levenberg-marquardt algorithm: Implementation and theory," in *Numerical Analysis*, ser. Lecture Notes in Mathematics, G. A. Watson, Ed. Berlin, Heidelberg: Springer, 1978, vol. 630, pp. 105–116.

[35] Embench Project, "Embench: Open benchmarks for embedded platforms," GitHub repository, 2019, accessed: 2026-08-22. [Online]. Available: https://github.com/embench/embench-iot

[36] PULP Platform, “riscv-gnu-toolchain: GNU toolchain for PULP and RISC-V,” GitHub repository, accessed: 2026-08-22. [Online]. Available: https://github.com/pulp-platform/riscv-gnu-toolchain

[37] ——, “Bender: A dependency management tool for hardware projects,” GitHub repository, accessed: 2026-08-22. [Online]. Available: https://github.com/pulp-platform/bender

[38] yttuncel, “Mismatch between simulation and FPGA emulation behavior,” GitHub Issue #274, pulp-platform/pulpissimo, 2021, accessed: 2026-08-22. [Online]. Available: https://github.com/pulp-platform/pulpissimo/issues/274

[39] J. A. Pienaar, A. Raghunathan, and S. Chakradhar, “MDR: Performance model driven runtime for heterogeneous parallel platforms,” in *Proceedings of the International Conference on Supercomputing (ICS)*, 2011, pp. 225–234.

A

Appendix 1: Ethical Aspects and Use of Generative AI

This thesis relies on open-source hardware and community infrastructure throughout, including Pulpissimo's Solderpad-licensed RTL and its public issue tracker for root-causing FPGA-specific defects. All modifications made in this thesis comply with that license and can be freely distributed or built upon. More broadly, the methodologies proposed here are motivated by lowering the barrier to meaningful architectural evaluation for academia and resource-constrained teams, an accessibility goal central to this work.

The generative AI tool, Claude, was used in a supporting and editorial capacity during the thesis, specifically for:

- Understanding and debugging tool usage (OpenOCD, GDB) when faced with issues.
- Support with scripting to automate the experimental design sweeps.
- Language editing and proofreading, including grammar corrections and sentence restructuring for clarity.

All AI-assisted text and code were reviewed, verified, and, where necessary, corrected by the author prior to inclusion in this thesis. The author takes full responsibility for the accuracy and originality of the final submitted work.