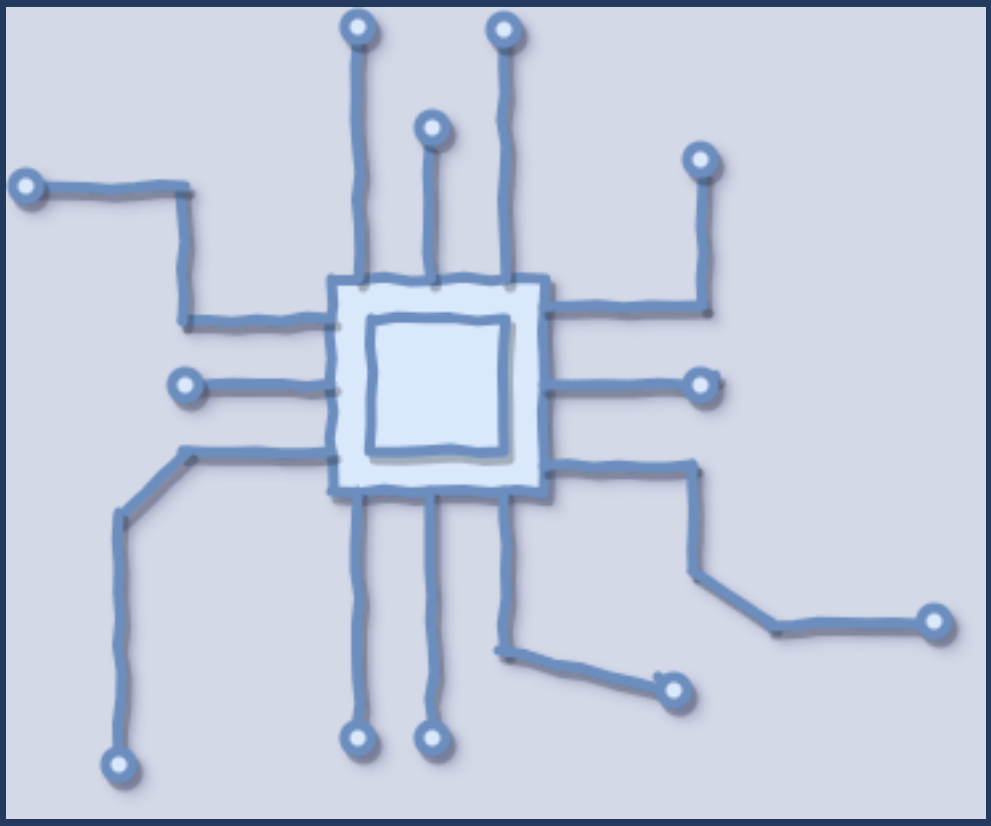
A Methodology for Improving gXR5 Speed and Accuracy

Karan Pathak



TU Delft

A Methodology for Improving gXR5 Speed and Accuracy

Master Thesis

by

Karan Pathak

to obtain the degree of Master of Science
(MSc) in Embedded Systems (ES)
at Delft University of Technology, The Netherlands

Thesis committee:

Chair:	Prof. Said Hamdioui, TU Delft
Supervisors:	Prof. Georgi Gaydadjiev, TU Delft
	Prof. Marina Zapater, HEIG-VD, HES-SO
	Dr. Giovanni Ansaloni, EPFL
	Prof. David Atienza, EPFL
	Prof. Said Hamdioui, TU Delft
External examiner:	Prof. Koen Langendoen
Place:	Computer Engineering, Faculty of EEMCS, TU Delft.
Project Duration:	November, 2022 - August, 2023
Student number:	5479614

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Faculty of Electrical Engineering, Mathematics and Computer Science Engineering
Delft University of Technology, The Netherlands

Dedicated to my parents,
Mukesh and Bharti

Preface

Computer Architects often walk the tightrope between performance, power and area while designing modern day processors. This daunting task is made even more challenging by short Time-to-Market requirements set by the clients. In light of these challenges, architectural simulators provide a much needed tool for the architects to gauge the impact of their innovations rather quickly. Arguably, use of such simulators is essential in avoiding a product recall due to the processor failing to deliver on performance/power requirements for the intended application. The thesis is in-line with the objective of identifying and addressing the bottlenecks in the RISC-V simulation ecosystem and contribute in the development of RISC-V infrastructure.

The intended objective of an architectural simulator is to capture the trend of the real hardware (i.e., performance improvement due to micro-architectural changes in real hardware should be eloquently captured by the simulator). A good simulator shall have high throughput (less simulation time) and should be easily re-configurable. The re-configurability of the simulator can be as fine as micro-architectural changes or as large as a new ISA being simulated. These attributes of speed and re-configurability come at the cost of accuracy. A high error in performance statistics of the simulator fails to engender confidence among the prospective users. Hence, validating performance of simulators against hardware is essential.

The thesis introduces the need for a full system architectural simulator for RISC-V processors followed by a brief, yet crisp review of the past attempts at making such simulators. The review is from the perspective of existing methodologies for performance validation of the simulators. The work also proposes a new methodology for validating system simulators. Although, the proposed methodology is generic and can be extended to other ISAs (such as ARM, x86, etc.), the target hardware chosen are RISC-V ISA based systems that span both commercially, IP protected processor as well as open-source processors widely adopted by the RISC-V community.

The work concludes by illustrating the future challenges to be addressed to make RISC-V simulation ecosystem vibrant.

Acknowledgements

"Vasudhaiva Kutumbakam"(The World is one family)
-Maha Upanishad

It's the journey that is worth cherishing and not the end. The experience of carrying out research at a university surrounded by the nature's calmness, the bright sun shining on the lake with blue sky, was surreal. This one-year long journey had ups and downs. The start was not the best, but the end seems to be connecting a lot many dots. The journey was full of excitement, sleep-less nights and happiness (of doing something 'significant'). Looking back, I feel satisfied and proud of what has been achieved.

The work could never have been carried out without the utmost support of my **parents**, their unconditional love and unfathomable belief in me (more of pride). I would like to thank my supervisors **Prof.Georgi Gaydadjiev** and **Prof.Marina Zapater** for showing me the right path. I consider myself lucky that I find the right people at the right moments in life. That also includes **Dr.Giovanni Ansaloni** for offering advises, providing me an enabling (and positive) environment and always being more than willing to help me. A big thanks to **Joshua Klein** for getting me started with the project and all off-the record conversations. You have made an impact in my journey for sure!

I would like to offer my sincere thanks to my supervisors **Prof.David Atienza** and **Prof.Said Hamdioui** for giving me the opportunity to work with them. My gratitude towards **Dr.Arjan van Genderen**, for helping me in navigating the treacherous bureaucratic procedures to have this (Inter-University) master thesis. A big thanks to my teacher for Embedded System courses **Prof.Koen Langendoen** and all my teachers who have made an impact in my life. I would also like to thank my cousin **Abhimanyu Kaushal** and his better half **Vaishally Bhardwaj**, for their kind support and bearing with my sense of humor. At last, my friend **Berkin** for all the good moments we had together and the late night "walk the talk" that helped me relieve the stress. My thanks to TU Delft and EPFL for sponsoring my research through **JvE Research Grant** and **SEMP scholarship**.

Contents

Nomenclature

AMAT: Average Memory Access Time

BHT: Branch History Table

BPKI: Branch per Kilo Instructions

BRAM: Block Random Access Memory

BTB: Branch Target Buffer

CHISEL: Constructing Hardware in a Scala Embedded Language

DPFMA: Double Precision Fused Multiply Accumulate

DSP: Digital Signal Processing

FP: Floating Point

FPGA: Field Programmable Gate Array

FS: Full System

gXR5: gem5 Extensions for RISC-V

HBM: High Bandwidth Memory

HPC: Hardware Peformance Counters

ILP: Integer Linear Programming

IMLI: Inner Most Loop Iterator

IntALU: Integer Arithmetic & Logic Unit

IPC: Instruction per cycle

LPDDR: Low Power Double Data Rate

M-PATH: Modulo Path history

MAC: Multiply and Add

MAPE: Mean Absolute Percentage Error

MBKI: Misses per Kilo Instruction

MBPKI: Mispredicted Branches per Kilo Instructions

MIPS/KIPS: Million/Kilo Instructions per second

MMU: Memory Management Unit

MP-BP: Multiperspective Perceptron Branch Predictor

MSHR: Miss Status Handling Register

PC: Program Counter

PPN: Physical Page Number

RAS: Return Address Stack

RR: Random Replacement (Cache Policy)

RTL: Register-Transfer Level

SE: System Emulation

SPFMA: Single Precision Fused Multiply Accumulate

TAGE_SC_L: TAGE, Statistical Correlator and Loop Predictor

VPN: Virtual Page Number

XSDB: Xilinx System Debugger

List of Figures

List of Tables

1

Introduction

1.1. Context

Computer Architects rely on simulators to evaluate the design options before fabricating the real hardware.These simulators can be classified broadly into two categories: Functional and Performance Simulators. The functional simulators (also called ISA-simulators) provide a qualitative test of the simulated hardware. Since these simulators abstract from the micro-architectural details, they have very high simulation speed. Examples of such simulators include Spike [1], QEMU [2], sim-safe model of SimpleScalar [3] etc. On the other hand, the performance simulators (also called Timing-simulators) model the micro-architecture of the hardware, thereby giving cycle-accurate statistics. Often, these simulators are event-driven (as opposed to clock-driven) to make up for the decreased simulation throughput. Examples of such simulators are gem5 [4], GEMS [5], PTLsim [6], etc.

Simulators can also be categorised into Application-specific and Full-system simulators. Application-specific simulators can execute synthetic micro-benchmarks. Poor simulation throughput, usually measured in KIPS/MIPS [7], restricts the use of large applications that require Operating System (and its libraries) to be run on these simulators. RTL simulations fall into this category of simulators. A full system (FS) simulator executes the user space applications atop filesystem and kernel, thereby enabling wider range of applications that can be run on the simulated hardware. gem5 stands as one the strongest open-source simulators that provides full-system capabilities, high reconfigurability as well as compatibility with other modeling tools such as Ramulator (modelling DRAMs) and McPAT (modeling Power). gem5 is a merger of two simulator infrastructures, namely m5[8] and GEMS[5]. It derives the memory system from GEMS and the CPU

models from the m5 network simulator. It supports full system simulation and system emulation by providing tunable system-level architectural models as well as processor micro-architectural models. These models are made ISA-agnostic by remapping ISA specific instructions into “op-classes”. The Figure 1.1 depicts the gem5 simulator with gem5 decoder enabling re-use of various models for different ISAs being simulated. The ease of reconfigurability makes gem5 one of the most versatile open source architectural simulator. Moreover, gem5 supports system emulation (SE) mode whereby the system calls are emulated on the host system and the results are delivered to the guest (simulated) system. This guest to host translations improves the simulation throughput but at the same time adversely affects the accuracy (i.e., it increases the disparity between performance statistics compared to the actual hardware) of the simulator.

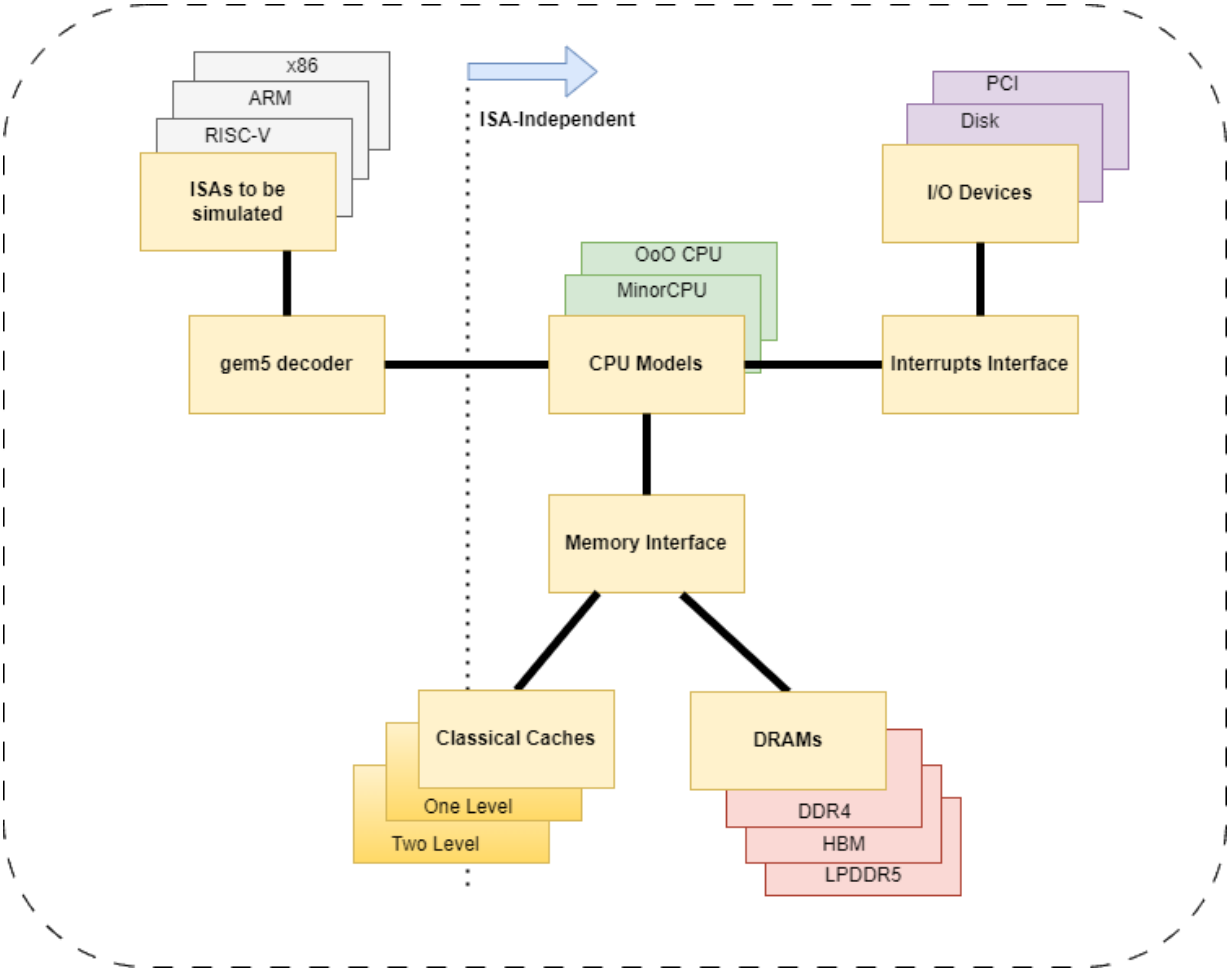


Figure 1.1: gem5 Simulator Infrastructure

1.2. Motivation

The RISC-V ecosystem has been witnessing tremendous push by the industrial players to accelerate adoption of RISC-V chips in IOT, mobiles, Data centres, automobiles etc [9]. Hardware prototyping is expensive and time taking and so, many industrial players (such as ARM) use full-system simulators to accelerate the chip development process[10]. These simulators have to be representative of the performance of the target hardware. A simulator validated for performance against an actual RISC-V hardware would be a defining contribution towards accelerating RISC-V proliferation. Moreover, the methodology for validating simulators have not been streamlined yet. The "**gem5 eXtensions for RISC-V**" [11], from now on **gXR5**, is a full-system simulator built on top of gem5. The objective is to bridge these research/industrial gaps by validating gXR5 against commercially launched Sifive Highfive Unleashed Freedom board [12] as well as RISC-V Rocket core [13] emulated on VC707 FPGA [14]. The Sifive Unleashed is a first Linux capable RISC-V system that is commercially available. The soundness of the proposed methodology to validate is illustrated by validating gXR5 against the IP protected processor, details of whose micro-architecture is not open source. On the other hand, rocket core is one of the most popular open source RISC-V hardware [15]. It has been chosen as a target hardware to prove the fidelity of the proposed methodology. gXR5 builds upon the gem5 simulator and hence, inherits the ISA-agnostic models such as CPUs, Caches, Interconnects, DRAM Controllers etc available in gem5. These behaviour models are written in C++ and made configurable via python bindings (Figure 1.2). Hence, simulation models can be configured via python scripts, facilitating fine changes in micro-architecture rather easily. Moreover, the python configuration scripts accept arguments via the command line to change the attributes of the models.

The conventional RTL test bench simulation of processors has been used in industry given its cycle-accurate performance statistics [16]. Unfortunately, the overhead of RTL design, development, and simulation for architectural exploration is large. In order to see the impact of minor change in the ISA (say adding vector instruction) on user-level benchmarks, extensive changes have to be made in RTL model of the processor core, peripheral systems,test bench, and even the compiler (or cross-compiler). Not to forget, RTL simulation has a very poor simulation throughput compared to This significantly restricts the utility of such RTL simulations for rapid SoC development.

Other simulators such as QEMU [2], SystemC [16], SPIKE [1], and OVPsim [17] have been widely used for simulating systems based on other ISAs (x86 and ARM). The OVPsim does not provide cycle-accurate results unlike gem5. A noteworthy competitor

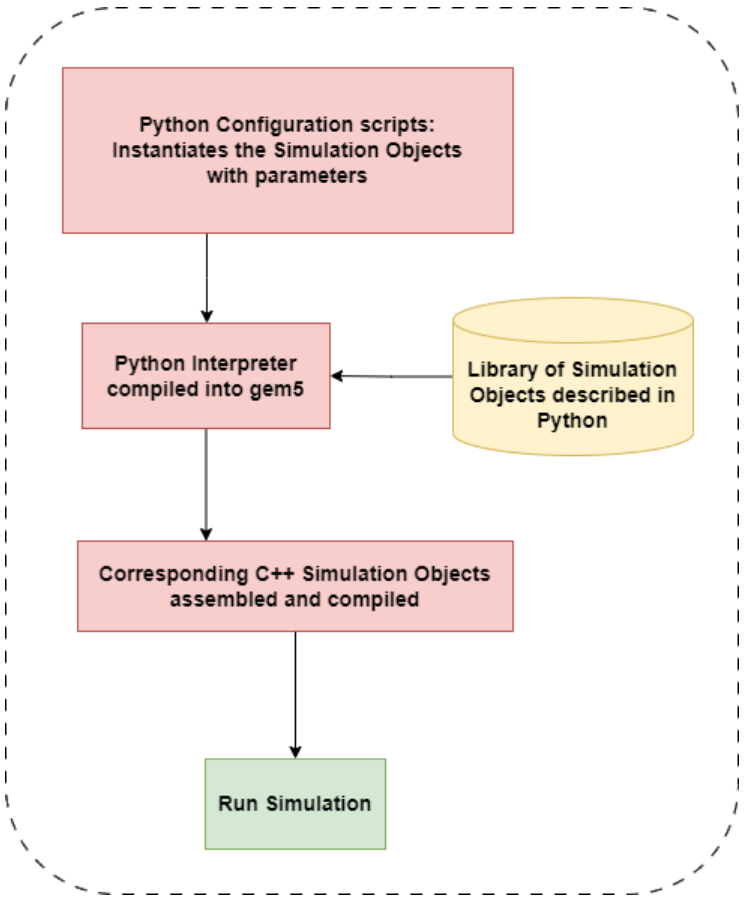


Figure 1.2: Configuring gem5 via python scripts

of gem5 is QEMU, which supports full-system simulation of RISC-V based platforms but only emulates system calls in the host platform, thereby losing greatly on accuracy compared to gem5. On the other hand, SPIKE only simulates the CPU core and caches, thereby offering far inferior functionality compared to gem5. The Figure 1.3 depicts the trade off between accuracy and speed of various simulators. gXR5 brings the best of both worlds, higher speed (compared to RTL Simulations) and higher accuracy (compared to functional simulators).

1.3. Contributions

This work proposes a new methodology that is accurate, fast and generic for validating gXR5. The accuracy is on par with existing full system simulators for x86 and ARM architectures. It is faster in terms of the validation process as it requires far less simulation runs to achieve similar performance accuracy compared to the existing methodologies. Moreover, one does not need to perform extensive hardware characterisation as only

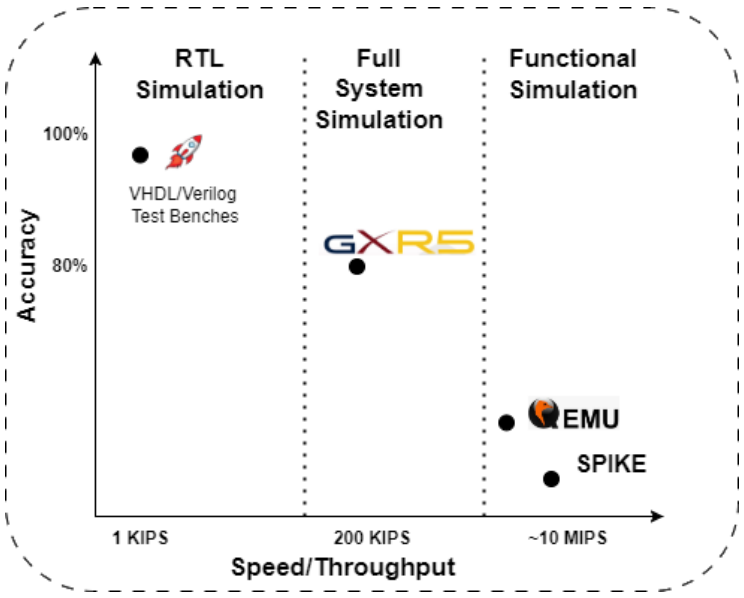


Figure 1.3: Comparison of RISC-V architectural simulators

Instruction and Cycle counts are used. Lastly, the methodology is generic and can be extended to other ISA based processors. The key contributions of the work are as follows:

1. a novel “component-level” calibration methodology for fine-tuning and validating full system simulators and demonstrated it’s advantages using gXR5 and two real RISC-V implementations;
2. the implementation and the validation of the first Gshare branch predictor model compatible with current multi-threaded gem5 CPU models;
3. the performance validation of open source gXR5 using selected SPEC CPU2017 benchmarks and real HiFive Unleashed SoC and Rocket Chip emulated on FPGA. We show our simulated results to be within 19-23% as compared to the two hardware targets above.
4. removed the need for licensed FPGA emulation to run user’s workload on Rocket Chip, making the Rocket Chip ecosystem completely open source.

The thesis makes the system design ecosystem truly open source. It provides validated open source simulator targeting open source hardware based on open source ISA (RISC-V).

1.4. Thesis Organisation

The thesis carries out a brief survey of various architectural simulators and sources of errors in simulators in Chapter 2. It then highlights the existing methodology of

micro-architectural level calibration using Hardware Performance Counters to validate simulators. It gives an estimate of the acceptable error in simulators used in industry/academia. Chapter 3 highlights the background work, including experimental setup, and detailed elaboration of selected gXR5 models. The micro-architecture of the open-source hardware (Rocket chip) is discussed only from the point of view of validation (with the focus on latency of different components). Chapter 4 is the heart of the thesis wherein the methodology is proposed and CPU model of the simulator is calibrated against the two target hardware. Chapter 5 discusses the performance results for selected SPEC2017 benchmark applications.

2

Literature Review

"There are three kinds of lies: lies, damned lies and statistics"
-Mark Twain/Benjamin Disraeli

Architectural simulators provide a platform for the computer architects to evaluate system performance quickly. A good number of these simulators have been built to serve pedagogical needs in universities such as ANT [18], CPU Sim [19], RM [20], etc. The chapter focuses on simulators that find themselves in research and industrial applications. It also provides a concise 'state-of-the-art' in performance validation of simulators. It is expected to give the reader an idea about the standard 'accuracy' of the simulators that are industrially relevant, despite having errors. The sources of errors that are common to all the architectural simulator have also been discussed. Finally, the existing methodologies for carrying out performance validation of simulators (especially full-system gem5 simulations) have been summarised.

2.1. Architectural Simulators

The section restricts itself to simulators capable of simulating the entire computer system. Multicore and multi-processor simulators such as Structural Simulator Toolkit (SST) [21], ZSim [22], Sniper [23] have been kept out of the scope as the focus of the work is on validation of single-core simulation. The objective of the survey is to situate the gem5 full-system simulator among the cohort of the available simulators, highlighting its importance for RISC-V ecosystem.

The various simulators can be classified based on the target metric, the method of

driving the simulation and interaction between simulation models, and complexity of simulation models. The following sections classifies the existing simulators.

2.1.1. Target metric/Figure of Merit

The architectural simulators can fall into following three broader categories depending upon the intended use or target metric.

- **Functional Simulators:** The Functional Simulators are essential in testing the models to be simulated before timing details are incorporated in the models. These simulators abstract from micro-architectural level details thereby making them computationally lighter. Hence, these are also called ISA-simulators. Examples of functional simulators include SimSafe model of SimpleScalar simulator [3], 'Atomic CPU' model of gem5 simulator [4], etc. Other functional simulators that support RISC-V ISA are Spike [1] and QEMU [2] . Similar functional simulator exist for GPUs, such as Barra [24] which is capable of running CUDA applications on modeled GPGPU.
- **Timing/Performance Simulators:** The Performance or Timing Simulators model micro-architectural details. Although, the granularity of modeling micro-architectural details vary across these simulators (and even within a simulator, such as different CPU models), the primary objective of these simulators is to serve as a tool to gauge and improve the actual hardware that has been modeled. RTL simulators (with test benches running micro-benchmarks) is one such simulator that models even the gate (or combinational logic) delays, and thus, comes out as the most fine-grained simulator (as far as abstraction in modeling is concerned). gem5 is one of the most prominent performance simulators. gem5 has been the focus of this study and has been described in detail in Section 3.2.3. DRAMSim [25] is a timing simulator for modeline memory system. It models memory controller behaviour for various DRAM protocols such as DDR3, DDR4, LPDDR and HBM. Another memory system simulator, the Ramulator [26] performs similar functionality but it is highly extensible and modular. Both DRAMSim and Ramulator are compatible with gem5 (and can be used with the gem5 CPU models to simulate the entire system). SiNUCA [27] is a simulator developed for modeling non-uniform memory accesses.
- **Power/Energy Simulators:** The power/energy simulators evaluate energy and power (and area in some cases) of the modeled hardware. McPAT (Multi-core, Power, Area, Timing) simulator [28] can give designers metrics such as energy-delay-area^2 product (EDA^2 P) and energy-delay-area product (EDAP). Though, current versions of McPAT has not been extended for RISC-V ISA. Hence, making

McPAT compatible with gXR5 is a promising research gap to be filled. One of the earliest Architecture-Power simulators include Wattch [29] that provides framework to optimize micro-processors for power consumption. There are a plethora of energy simulators. HotSpot [30] models the die and the package as circuit of resistance/capacitance. These models can then be used at the architectural level for power modeling of SoCs.

2.1.2. Scope of target

The simulators can be classified into Full-system or System Emulation depending upon how the system calls are handled by the simulator.

1. **Full-System Simulators:** A Full-System (fs) simulator is one that is capable of executing user space applications on top of filesystem atop kernel. The kernel resides in the simulated hardware which runs on the host hardware. One of the earliest and arguably most popular of its time was SimOS fs-simulator capable of simulating MIPS CPU with memory virtualisation by MMU [31]. Sunflower [32], PTLsim [6] and MARSS-x86 [33] , MARSS-RISCV belong to this category. However, the PTLSim and MARSS-x86 have been designed for simulating 64-bit x86 system.
2. **System Emulation Simulators:** System Emulation simulators are capable of executing applications that require OS services/libraries by bypassing the system calls to the host system. These simulators suffer from high error when system calls make up significant proportion of the workload. QEMU [2] and gem5 system emulation are examples of this type of simulator.

2.1.3. Input to Simulators

The simulators can be classified into trace and executable driven based on the type of application being run.

- **Trace driven Simulators:** The trace driven simulator uses a pre-compiled set of instructions. The benchmarks are compiled on hardware and the trace of instructions recorded is then fed to the simulator. This leads to the simulator not modeling mis-speculation-logic (eg. branch mis-predictor). Though attempt have been made to incorporate extra logic implementing the mis-speculation path. Since the trace is collected from executing benchmark on another system, the accuracy of the simulator suffers. Shade is one of the earliest trace-driven ISA-simulator. Other such simulator includes MASE [34] which is built on top of Simplescalar toolset [3] and combines timing and performance model for a single core.

- **Executable driven Simulators:** Executable driven Simulators simulate the actual hardware, including the mis-speculations and squashing logic to restore the state of the system/processor. Hence, they are much more accurate but at the cost of complex code/logic of models and decreased simulation throughput. Simplescalar and SESC [35] fall into this category of simulators.

2.1.4. Driving agent of Simulation

Lastly, the simulators can be clock or event driven. These two types have been described below:

- **Clock driven Simulators:** The clock-driven simulators evaluate the modeled system at every rising/falling edge of the clock. RTL simulation that is highly accurate (and computationally intensive) is an example of clock-driven simulations. The impracticality of such simulators in executing macro benchmarks led to the development of event driven simulators.
- **Event driven Simulators:** In these type of simulators, the clock runs in background but the simulator evaluates the system only when a event takes place. Often these simulators are modular and hence, the modules interact by way of events. There is an event handler/scheduler that schedules/re-schedules the events sitting in an event queue. Simulating at the events significantly increases the simulation throughput without losing much on accuracy. SESC [35], gem5 [4], SST [21] are examples of event-driven simulator designed for MIPS ISA. The challenge arises in parallelizing the simulations (using OpenMP/MPI) as the events are sequential. However, considerable parallelism can be achieved in (sequential) event-driven simulations by using SimPoint [36] to parallelize the benchmark applications (such as SPEC2017) based on statistical sampling.

2.2. Simulation Errors

The Error in IPC and Execution time of simulator (Vs. target hardware) remains one of the most important metric, apart from the simulation throughput (measured in KIPS/MIPS) [22], [37], [38]. High-level computer architects often rely on execution time of simulators to gauge the run-time of application on real hardware. The hardware designers use IPC to measure improvements in (modeled) hardwares. The simulation throughput reflects the practicality of using simulators and is used to compare simulators. Attempts have been made to streamline the methodologies to reduce this error. Before proceeding to

the following section that describes these methodologies, it is important to discuss and distinguish the source of errors across simulation platforms. The sources of errors in the simulator can be placed into following three categories:

- **Abstraction Errors:** The developer makes a conscious decision of choosing the level of abstraction while modeling the actual hardware. The higher the abstraction, the less computationally intensive would be the simulations. Abstraction comes at the cost of accuracy. For example, Integer Division (by convergence) Functional unit may be modeled with a fix latency irrespective of the operands being operated upon. The hardware unit will have different latency for division operation depending upon how quickly the denominator converges to value of one.
- **Modeling Errors:** These errors are due to the designer failing to understand the functionality of the actual hardware being modeled. For example, the data cache access may be taking two clock cycles in the actual hardware but has been modeled with one clock cycle latency.
- **Specification Errors:** The specification errors are the most challenging of all. These represent errors due to the lack of knowledge of the architectural details of modeled hardware . A common reason for this is the behind the veil micro-architecture of commercial boards.

Needless to say, all the existing performance validation methodologies target the specification errors while validating for IPC/Execution time. The modeling errrors can be resolved by doing the functional validation. The simulators are validated against commercial ARM, x86 ISA compliant boards/processors. The following section discusses these methodologies.

2.3. Validation Methodologies

The earliest methodology for validating architectural simulators can be described as “validation by inspection” [39] wherein multiple simulation runs would lead to enough performance data that could be used to reduce error in the simulator. Gutierrez et al. validate Out-of-order (OoO) CPU model of gem5 against ARM A15 core. They use selected (11 out of 29 applications) SPEC2006 [40] benchmarks with train inputs set to calibrate for gem5 for run time against the actual hardware. Since, they rely on replicating the actual micro-architecture, they reduce the modeling errors significantly. They achieve a mean run time error for selected SPEC applications of 13% and for selected (7 out of 13 applications) PARSEC benchmark, an error of 11% and 12% for single and dual cores respectively. In more recent work Qureshi et al [41] tune and validate performance

of in-order and OoO cores against a real ARM JUNO platform developed by ARM (2015) with a mean absolute error (MAE) below 4 %. The methodology can be at best described as trial and hit method or validation by empirical inspection. Similar work has been done by [42] wherein they modify Out-of-Order (OoO) CPU model of gem5, making the pipeline in-order. This new CPU model is validated against a single and dual core leading to just 8 % run time errors (for single threaded ARM Cortex-A8, single and dual threaded ARM Cortex-A9) for selected (10 out of 13) PARSEC benchmarks [43]. Butko et al [44] calibrated GEMS for the ARM Cortex-A9, bringing the error between 1.39 % and 17.94% for SPLASH-2 [45], ALPBench [46] and STREAM [47] benchmark applications.

Desikan et al. [48] have extended SimpleScalar toolkit for simulating Aplha ISA. They validate this simulator (SimAplha) against alpha 21264 processor of DS-10L Workstation. They propose the methodology of using micro-benchmarks to calibrate individual models of the simulator. These microbenchmarks are divided three categories - “C ”(Control), “E” (Execute) and “M” (Memory):

1. C micro-benchmarks: These stress the front-end of the processor (Instruction (pre)fetches, line predictors, way predictors, branch predictor, etc.) They are further categorised into control-conditional (C-C), control-recursive(C-R), control-switch (C-S) and complex-control (C-O).
    - C-C: Implement simple if-then-else construct.
    - C-R : Tests the indirect jumps by having recursive function calls.
    - C-S: Implements 10-way switch case statements within a loop.
    - C-O: It is a hybrid of C-C and C-S, which loops over if-then-else statements.
2. E micro-benchmarks: These test the functional units. They are categorised into :
    - Execute-independent: They add the index variable to eight independent, integers stored in registers. They do it 20 times each within a loop.
    - Execute-float-independent: They have similar functionality as execute-independent except that they operate on floating point numbers.
    - Execute-dependent: They implement (register) dependent chain of operations within a loop.
3. M micro-benchmarks: These are categorised into following subcategories:
    - Memory-independent: These repeatedly execute the independent loads thereby testing the L1 Data cache bandwidth.
    - Memory-dependent: These test the L1 Data cache latency by walking a linked list.

- Memory-L2 and memory-memory micro-benchmarks are similar except that the working set size is kept large enough to intentionally cause L1/L2 cache misses.

The IPC error for these benchmarks are brought down from 74% to 2% for the above benchmarks. The validated simulator has less than 20% error for macro-benchmarks derived from SPEC2000 [49] benchmark suite.

Alves et al. [27] follow a similar approach by designing SiNUCA micro-benchmarks [50] for validating the cycle-accurate, trace-driven SiNUCA (Simulator for Non-Uniform Cache Accesses). These micro-benchmarks are classified into four categories, Control benchmarks (comprising of Control Conditional, Control Switch, Control Random and Control Complex), Execute benchmarks, Dependency benchmarks (stressing forward dependency between instructions) and Memory benchmarks. The simulator has been validated against Sandy bridge processor with (geometric) mean error in IPC of 6% for the above micro-benchmarks.

Akram et al.[51] use the SiNUCA micro benchmarks to validate gem5 against Intel (x86) i7-Core (Haswell micro-architecture). Additionally they use Hardware Performance Counters (HPCs) to observe the micro-architectural events in the actual hardware. The Pearson's coefficient [52] is used to find the correlation between error and the micro-architectural event. Pearson's correlation coefficient can capture the strength of correlation as well as the direction of error (positive correlation implies that the simulator is overestimating the performance of micro-benchmarks). They reduce the error in the micro-benchmarks to 6% and also implement a loop predictor in gem5. However, the validated simulator has not been tested by running macro-benchmark suite. Similar work (using HPCs and correlation coefficients) have been carried out for calibrating Out-of-Order (OoO) CPU model in gem5 against ARM Cortex-R8 CPU.

Recently, an attempt has been made to streamline the design of these synthetic benchmarks used to calibrate the simulator models. Huppert et al. [53] have designed micro-benchmarks for calibrating memory hierarchy using simplistic technique of 'Data-pinning'. The working set size is made to reside at different memory levels to calibrate access latency using HPCs. Unfortunately, the methodology can not be extended to calibrate other models (especially as complex) as CPU.

Section 4 proposes and implements the new methodology. The methodology reduces the specification errors in simulator by 20% just by calibrating the CPU model. The past attempts for validating simulators have been summarised in the Table 2.1.

Validated Simulator	Target Hardware	Training Set Error	Test Set Error
SimAlpha	DS-10L workstation (Compaq Alpha 21264 processor)	Synthetic microbenchmarks IPC Error of less than 2 %	selected macrobenchmarks derived from SPEC2000 suite IPC Error of 18 [48] %
SiNUCA	Sandy bridge processor	Synthetic microbenchmarks IPC Error of 9 %	SPEC2006 suite having an IPC Error of 19% [27]
gem5	ARM Versatile Express TC2 development board	SPEC and PARSEC	SPEC and PARSEC Run Time Error of 13 % and 11 % [39]
gem5	Cortex-A53 core of MediaTek Helio X20 SoC	Synthetic microbenchmarks	SPEC2006 suite IPC Error of 20 % [53]
gem5	Arm R8 CPU	Embench workload CPI Error of 13 %	No macrobenchmarks were run [54]
gem5	Intel Core-i7 (Haswell Microarchitecture)	Synthetic microbenchmarks IPC Error of 6 %	No macrobenchmarks were run [51]
gem5	ARM Cortex-A8 (dual core, Snowball SDK), ARM Cortex-A9 (single core,BeagleBoard-xM SDK)	PARSEC	Selected PARSEC benchmarks (with simsmall input set), Run time Error of 8 % for both [42]
GEMS	ARM Cortex-A9 (Snowball SKY-S9500-ULP-C01)	SPLASH-2, ALPBench, STREAM	same as test set, Run time Error of 1.39% to 17.94 % [44]

Table 2.1: Past simulator validation efforts with reported errors for benchmarks.

3

Preliminary Work

This Chapter delves into the the details of gXR5 models along with the simulation set-up. It gives an overview of the simulator, CPU models, memory system, etc. The micro-architecture of the two target hardware, namely Sifive Highfive Freedom Unleashed board and UC-Berkley's Rocket core has been dealt in detail.

3.1. gem5 Extensions for RISC-V: gXR5

gXR5 is a linux capable full-system simulator for RISC-V ISA based uni-core platforms. It is an event-driven, full-system performance simulator that builds on gem5 and so, it inherits all the models (CPU, caches, DRAM, TLBs, etc.) of gem5. Akin to gem5, it runs on top of the host OS and hardware (usually a different ISA). Figure 3.1 depicts the gXR5 running benchmark applications atop (guest) OS and simulated (guest) hardware, atop (host) OS and (host) hardware.

The simulator comes with a python interpreter that helps configure and instantiate the corresponding C++ models as discussed in Chapter 1. The simulator is run in full system mode by configuring (and connecting) the built-in processor, caches and memory models with the Simple Board components.The built-in UART model is used to interface an external terminal and PCI host model is used to interface the storage to the rest of the system via the the Platform Level Interrupt Controller (PLIC). The PLIC interfaces with the CPU to assign the interrupts to the core whereas the Core Local Interrupt Controller manages the software and timing interrupts of each individual core. Figure 3.2 provides a high level view of the built gXR5-full system simulator. The CPU model used is an in-order MinorCPU model that is interfaced with classical caches and Main memory by buses. Since gXR5 supports SV39 virtualization, the addresses undergo virtual to

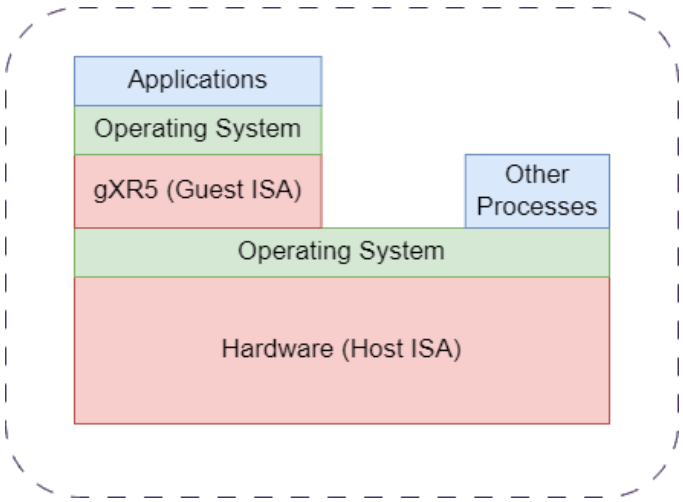


Figure 3.1: High-level view of gXR5 running on host system

physical translations via walkers or translation buffers (TLBs) before accessing the L2 cache and main memory. It is important to dive into the implementation details of these C++ models. Hence, the following section describes the micro-architectural details of the CPU,caches and DRAM models.

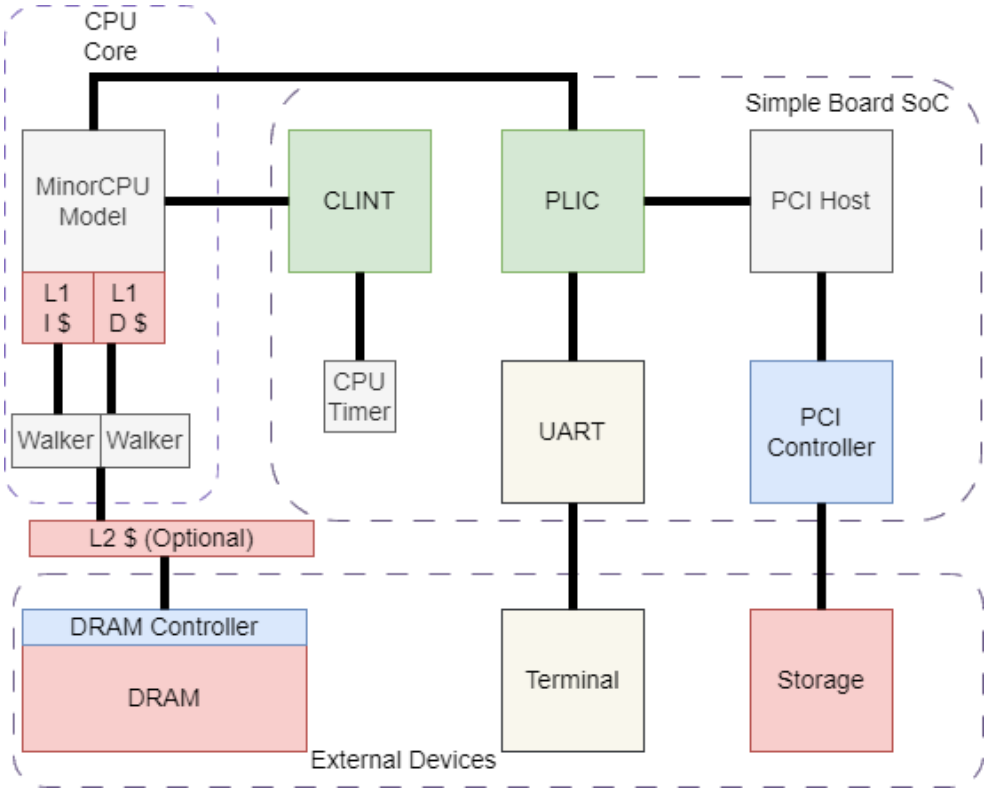


Figure 3.2: Simplified model of gXR5 full system simulator [11]

3.2. gXR5 Models

The basic building blocks of gXR5 are the Simobjects. A Simobject is a C++ wrapped object that can be configured and interfaced via the python configuration scripts. Almost all objects in gXR5 (such as CPU, caches, TLBs, branch predictor, functional units, etc) inherit from the Simobject. The Simobjects derive from the class Eventmanager which implements functionalities of scheduling/re-scheduling and waking up the events.

3.2.1. CPU Model

gXR5 comes with four in-built CPU models- Minor (in-order), Out-of-Order (O3), AtomicSimple and TimingSimple CPU models. The AtomicSimple and TimingSimple CPU models are single cycle CPU models (without pipeline). MinorCPU models derive from the BaseCPU model that implements the basic functionalities such as setting up a fetch request, handling pre-execute setup, handling post-execute actions, and advancing the PC to the next instruction. MinorCPU model is an in-order four stage pipelined CPU model with following stages :Fetch1, Fetch2, Decode and Execute. These stages are connected through input buffers that hold the instructions in case of a stall. The instructions traverse through to the next stage of the CPU only when the subsequent stage's input buffer has space to accommodate the instruction. Figure 3.3 depicts Minor CPU model with the four aforementioned stages.

Stage-I: Fetch1

The Fetch1 stage fetches the instruction from the L1-instruction cache and passes it onto the Fetch2 stage. Fetch1 unit tags the cache line fetched to enable Fetch2 unit to distinguish between sequential fetches (pc+4) and branched fetches (Target PC). The branched fetches are indicated by ”change of stream” signal communicated by Fetch2 (during branch predictions) and Execute stages (during branch mispredictions). Upon receiving the ”Change of stream” signal Fetch1 unit starts fetching from the new program counter which is fed by Fetch2 stage (or by Execute stage incase of branch mis-prediction). In case of contention between Fetch2 and Execute stage ”change of stream” signal, the Execute stage overrides. The Fetch1 stage comprises of two queues, namely, the Request Queue and the Transfer Queue that keeps track of the requests made to the caches (through the Address Translation Buffer) and responses received, respectively. The instructions are fetched from memory only if it can reserve a space in input buffer to Fetch2 unit, justifying the need to have Request and Transfer Queues.

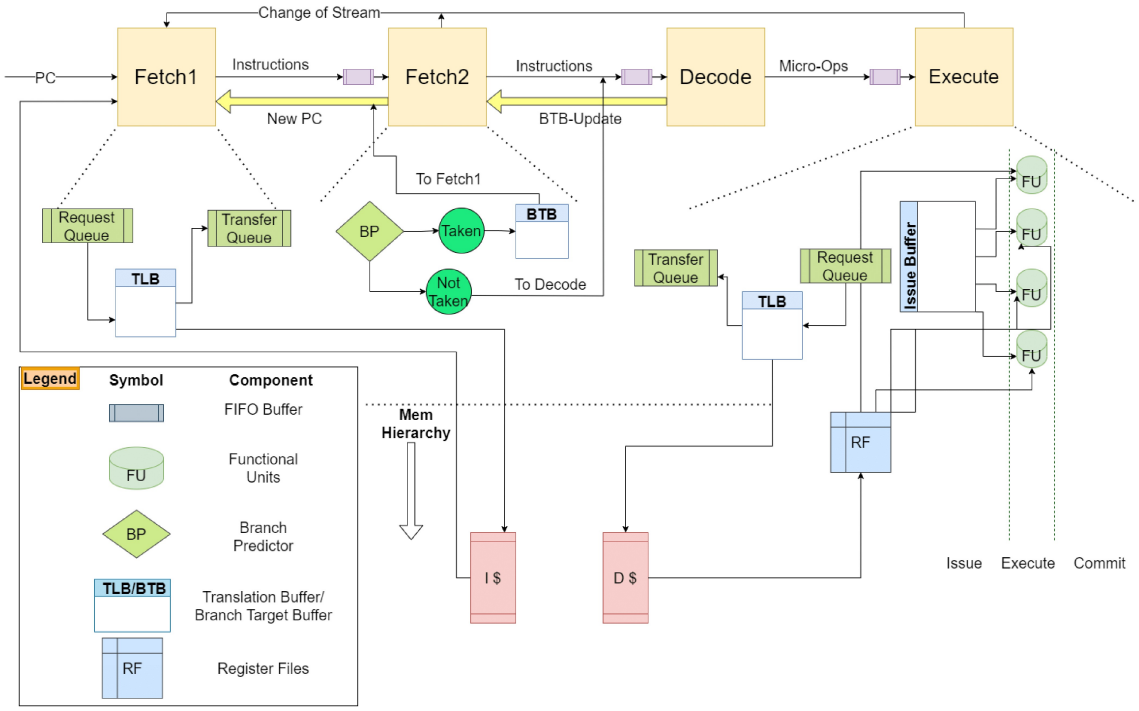


Figure 3.3: Simplified MinorCPU model in gXR5

Stage-II:Fetch2

The Fetch2 stage comprises of a Branch Predictor Unit. There are numerous built-in branch predictor models in gXR5 such as Tournament Bi-mode, Multiperspective Perceptron, TAGE branch predictor, etc. The branch predictor can be chosen and configured into the system via the python configuration script. The Branch predictor either signals ”change of stream” (while branch is taken) or feeds the Decode stage with a sequential instructions (while branch is not Taken). Fetch2 stage also performs a sanity check on stream number (tags attached by Fetch1 unit) and discards the instructions having different sequence number from that of predicted sequence number. This way the input buffers are flushed in-case of branch prediction/mis-prediction. The instructions are packed into vectors and sent to the next pipeline stage. The Fetch2 unit updates the Branch Target Buffer (BTB) and Branch History Table (BHT) with the updates received from Decode Unit.

Stage-III: Decode

The Decode stage breaks the instructions into micro-ops and packs these into vectors before passing them to the input buffers of the Execute stage.

Stage-IV: Execute

The input buffer to the Execute stage stores the micro-ops. The Execute stage itself is divided into issue, execute, and commit sub-stages. The micro-ops rest at the issue stage if the corresponding functional unit is pre-occupied. Once the functional unit is available, the micro-op is popped from the issue buffer and executed. The commit stage marks the end of instruction life-cycle. The size of the output vector (number of micro-ops packed) from Decode stage and the size of the input buffer to Execute Unit are intricately related and hence should be tuned simultaneously, otherwise it will lead to stalls. The data fetch requests to memory go through Request and Transfer Queues (similar to Fetch1 Unit). A separate Store buffer ensures that the load and store operations do not hinder each other.

Pipeline

The pipeline class has an “event” associated with it which is scheduled for every clock tick. The event triggers the method “evaluate()” that calls evaluate method on each stage in reverse order. The order ensures that when the next stage of the cpu stalls, the previous stages too shall stall.

Listing 3.1: Pipeline-MinorCPU model

```
void
Pipeline::evaluate()
{
    cpu.tick();

    execute.evaluate();
    decode.evaluate();
    fetch2.evaluate();
    fetch1.evaluate();
}
```

3.2.2. gxR5 - Memory system

The memory system in gem5 comprises of memory objects (caches, DRAM, etc), ports and packets that are transferred between memory objects. All memory objects inherit from the MemObject class which implements basic functionality such as returning the slave/master port based on name and index. The Memobjects are connected to each other and the CPU via the ports. The ports come in pairs - master and slave. CPU can have more than one master ports interfacing to main memory. Similarly, the DRAM can

have multiple slave ports feeding multiple memory objects (caches) and CPU. The ports have send and receive functions that implement sending and receiving (at the peer side) the packets.

To illustrate the flow of data from CPU to main memory, a request object is created with following attributes :

1. Virtual address (in case of virtually addressed caches/memory) or physical address.
2. Data size.
3. Time stamp of request created.
4. Thread ID creating the request.
5. The Program Counter creating the request (say, Load/Store instruction)

The request transcends the different memory objects/ports and is broken up in packets. A request travels from sender to the ultimate receiver and back, whereas the packets are exchanged between two objects. The packet encapsulates the following (and not all) important attributes:

1. The address: This is derived from the address specified in the request object.Hence, this could be physical or virtual address. In case of a cache miss, this address will be different from the address specified in the request object.
2. The Data size: In case of a cache miss, the entire block needs to be fetched from the main memory to the cache. This is the only scenario when the data size of packet would be different from the requester objects data size.
3. A pointer to the data being manipulated.
4. A flag to indicate success of data transfer and otherwise.

The memory objects and CPU can be connected and configured by python scripts. The following sections give an overview of the built-in memory objects used in gXR5, namely, Classical Caches and DRAM Models.

Classical Caches

The classical caches are non-blocking caches, i.e., they support hits under a miss by hosting a Miss Status Holding Register (MSHR). The various attributes of caches such as cache and block size, associativity, cache block replacement policy, write back/through policy, etc. are parameterized and can be configured by the corresponding python script for caches. The classical caches come with just MOESI coherence protocol and support snooping for maintaining cache coherency. In gXR5, the snooping requests have been made different from ‘normal’ memory requests. The snooping requests traverse

horizontally and up the cache-hierarchy whereas the memory requests go down the cache/memory hierarchy. Similarly, the snooping responses travel horizontally and vertically down on the same route as the snooping requests.

The caches are interfaced to the main memory via the coherent bus (cross-bar). The cross-bar has slave ports on the CPU side and master ports on the main memory side. The average memory access time (AMAT) for caches in gXR5 can be mathematically represented by the following equations.

$$Average\ Memory\ Access\ Time\ = (Hit\ Time)_{L1} + (Miss\ rate)_{L1} * (Miss\ Penalty)_{L1} \tag{3.1}$$

where

$$Miss\ penalty_{L1} = (Hit\ Time)_{L2} + (Miss\ rate)_{L2} * (Miss\ Penalty)_{L2} \tag{3.2}$$

Where, in gXR5

$$Hit\ time = f(response\ latency, data\ latency, tag\ latency) \tag{3.3}$$

$$Miss\ penalty = f(mshr\ queue\ size, Hit\ time\ of\ lower\ level\ memory) \tag{3.4}$$

$$Miss\ rate = f(associativity,\ clusivity,\ replacement\ policy) \tag{3.5}$$

The hit-time, miss penalty and miss rate can be fine-tuned to bring the AMAT (of instruction and data fetches) of simulated memory system closer to the actual memory system.

The miss penalty associated with last level cache (L2) is dependent on the DRAM model (access latency). The precise latency for load/store instruction has been summarised by Wang et al. [54] as follows:

$$Lat_{L1D} = OpLat_{Mem} + Max(TagLat_{L1D}, DataLat_{L1D}) \tag{3.6}$$

The $OpLat_{Mem}$ represent the latency of the Load/Store Unit (LSU). The tag comparison and the data access happen in parallel and hence, a maximum latency of both is added to the LSU latency to get the hit latency of L1 Data Cache represented as Lat_{L1D}. Similarly, in case of L1 miss, the total latency comprises of LSU latency, tag look up latency, X-bar latency, access latency of L2 cache and the latency encountered in the return path ($RespLat_{L1D}$).

$$Lat_{L2} = OpLat_{Mem} + TagLat_{L1D} + RespLat_{L1D} + MemLat_{L2} + Lat_{Xbar} \tag{3.7}$$

The following section presents an overview of the DRAM controller models available in

gXR5/gem5.

DRAM models

gXR5 comes with in-built DDR3, DDR4, Low Power DDR (LPDDR) and High Bandwidth Memory (HBM) models that derive from the class ‘DRAMCtrl’. The class ‘DRAMCtrl’ implements basic functionality associated with DRAMs such as ranks, banks, DRAM packets. It also holds data structures (queues) for tracking read/write requests received by the DRAM. Few of the DDR models available and their peak transfer rates have been summarised below.

1. DDR3_1600_8x8 with peak transfer rate of 12.8 GBps. The DRAM operates at 1600 MhZ.
2. DDR3_2133_8x8 with a peak transfer rate of 17.0 GBps. The DRAM operates at 2133 MhZ.
3. DDR4_2400_16x4 with a peak transfer rate of 19.2 GBps. The DRAM operates at 2400 MhZ.
4. DDR4_2400_8x8 with a peak transfer rate of 19.2 GBps. The DRAM operates at 2400 MhZ. In comparison to the previous DRAM model, this has 8 banks each 8 bit wide.
5. LPDDR2_S4_1066_1x32 with a peak transfer rate of 4.3 GBps.
6. HBM_1000_4H_1x64 with a peak transfer rate of 8.0 GBps.

The default scheduling policy implemented by the DRAMCtrl is ‘frfcfs’ i.e first come first served with row-open hit (first row) being prioritised. The default configuration also uses single bank (group) in a rank with a single channel to the memory.

gXR5 uses the classical caches with one and two level hierarchy along with in-built DDR3 and DDR4 DRAM models. The default configuration of these models are used to replicate the DRAMs used in Sifive Unleashed board and Rocket system emulated on FPGA, leaving calibration of the memory hierarchy as future work.

The focus of work is on calibrating the Minor CPU model. Branch predictors are an indispensable part of modern processor and the biggest source of performance discrepancy for simulators. The following section gives an introduction to the types and micro-architecure of the in-built branch predictor models available in gXR5. The models are desrcibed breifly as a design space exploration has to be carried out to choose a model that is closest to the actual branch predictor used in hardware.

3.2.3. gXR5-Branch Predictor Models

The Base class for all branch predictors models in gXR5 is BPredUnit that inherits from the SimObject class. Some of the basic and important public member functions of the 'BPredUnit' class are as follows:

1. *bool lookup (ThreadID tid, Addr instPC, void *&bp_history)* : The function looks up the branch prediction decision based on PC and branch history. Since the MinorCPU model is multithreaded, each thread (tid) has its own history, PC, prediction, etc. (the 'Context'). The function returns either true or false based on the prediction 'taken' and 'not-taken' respectively.
2. *void update(ThreadID tid, Addr branchAddr, bool taken, void *bpHistory, bool squashed, const StaticInstPtr & inst, Addr corrTarget)*: The function updates the branch history (actual) decision. This could be local or global history depending upon the type of branch predictor implemented. The function also handles 'squashes' incase of branch misprediction.
3. *void btbUpdate(ThreadID tid, Addr branchAddr, void * &bpHistory)*: The function updates the BTB with the target address.
4. *TheISA::PCState BTBLookup (Addr instPC)*: The function looks for given PC in the BTB.
5. *void uncondBranch(ThreadID tid, Addr pc, void * &bpHistory)*: The function is called for unconditional branches and updates the bpHistory with (always) 'taken'.

The other member functions implement squashing, logic for valid BTB addresses in case of BTB update, etc. The current gem5 repository (that gXR5 inherits from) has the following MinorCPU compatible branch predictors:

1. Local Branch Predictor
2. Bi-Mode Branch Predictor
3. Tournament Branch Predictor
4. TAGE Branch Predictor (with Statistical Correlator and Loop Predictor)
5. Multi-perspective Perceptron Branch Predictor

The following section briefly describes these branch predictor models.

Local Branch Predictor

The Local branch predictor predicts the branch based on local history of each branch. Each branch address is used to index into the local counters which hold the history of a

particular branch. It uses a 2-bit (default) saturating local counter. This is perhaps the most basic branch predictor implemented in gXR5.

Tournament Branch Predictor

The branch predictor comes with a single global history register that is updated for every take/not taken (actual) outcome of the branch. Along with the global history, the local history is maintained for each branch address (PC) with a configurable saturating counter. There also exists choice counters that decide between global and local history based prediction. The choice counters are indexed by the global history (masked by 'choicehistorymask'). The choice counter is incremented if the outcome matches the global history based prediction and decremented otherwise. A pre-defined threshold is set for choice counters, above which the global history predictor is chosen.

Bi-Mode Branch Predictor

The Bi-mode predictor comes with two direction predictors and a choice predictor, alike the aforementioned Tournament Predictor. The fundamental differentiating factor being that the direction predictors are indexed by hash of branch address (PC) and global history register.

TAGE Branch Predictor

The TAGE branch predictor [55] implemented in gXR5/gem5 follows the TAGE branch predictor proposed in Branch Predictor Championship. It comes in two flavors-16KB and 64KB size. The default implementation has 8-component TAGE predictor with seven T-Tables and a basic branch predictor (tag-less, bi-mode indexed by PC). The T-Tables are indexed by hashing the branch address (PC) and the variable (geometric progression series) global history. The prediction by basic predictor is chosen only incase of Tag miss in all the T-Tables. The implementation is based on the predictor proposed at the Branch Predictor Championship.

Multi-perspective Perceptron Branch Predictor

The Multiperspective Perceptron branch predictor [56] (MP-BP from now on) adds more 'perspectives' beyond the conventional perspectives (discussed above such as local and global history) to make direction predictions. These include the PATH history, Inner-Most Loop Iterator (IMLI), Modulo-PATH history (MPATH), etc. Trained weights are attached to all these perspectives that are used to index the table of counters.

TAGE-Statistical Correlator and Loop Predictor Branch Predictor

The TAGE-Statistical Correlator and Loop Predictor branch perdictor (TAGE-SC-L BP [57], from now on) comes with a TAGE branch predictor backed by statistical correlator and loop predictor. The statistical correlator makes up for the poor performance of TAGE predictor in case of branches that have only a small bias towards a direction, but are not strongly correlated with the history path. The TAGE performs poorer than normal PC indexed prediction in such cases. Hence, the statistical correlator decides whether to invert the direction predicted by TAGE predictor or not. The loop predictor simply identifies regular loops with a fixed number of iterations and makes predictions.

The gXR5 ISA-agnostic models discussed above can simulate a wide range of SoCs. The models are aptly abstracted from finer details (like gate and combinational delay) and hence, save a lot of computation power thereby increasing simulation throughput. The functional units too follow the same principle. The functional units are modeled as black boxes with tune-able latency such as ‘Issue Latency’, ‘Operation Latency’, etc. The micro-architectural details such as type of adder (say ripple-carry, carry-skip, etc.), multiplier (say higher radix, iterative, etc.) can be implemented by changing the aforementioned latency attributes. Nevertheless, it is important to delve into the micro-architectural details of the target hardware in order to reduce the performance disparity between simulator and actual hardware.

3.3. Target Hardware

The following section describes the micro-architecture of the target hardware. gXR5 is being validated against two target hardware, namely, Sifive Highfive Freedom Unleashed[12] and Rocket Chip [13] emulated on VC707 FPGA [14]. The Sifive Unleashed is the first Linux Capable RISC-V board that is commercially available. On the other hand, RISC-V Rocket Chip is one of the most popular open-source processor available. Figure 3.4 uses git activity, citations , tape outs and FPGA emulations of open source hardware including CVA6, BOOM (out-of-order) and Shakti processor. Rocket stands out as the most promising candidate to test the fidelity of the proposed methodolgy.

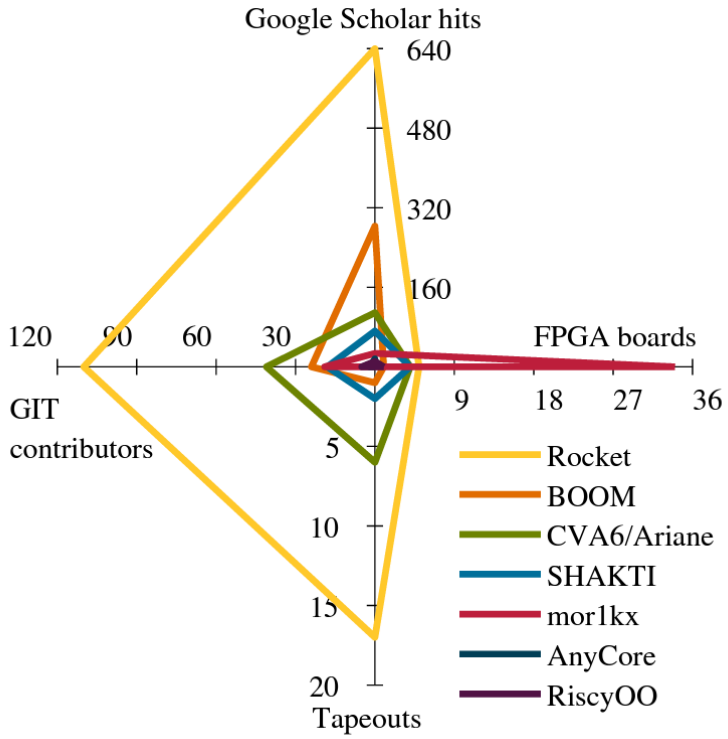


Figure 3.4: Comparison of Open-source RISC-V hardware[58].

3.3.1. Sifive Highfive Unleashed

The Sifive Highfive Freedom Unleashed is a quin-core, Linux-capable SoC. It has four FU540-C000 in-order CPU capable of hosting Linux. The other core, the S51, supports Real-Time Operating Systems. The FU540-C000 (also referred to as U54) is a RV64GC core. The figure 3.5 provides a high-level view of the Unleashed board. We base our simulated model and validation effort on single-core workloads executing on the U54 core. The following section divulges the micro-architectural details of U45 core available in the public domain.

Micro-architecture

The U54 core has a high performance single-issue 64-bit execution pipeline, sustaining a peak throughput of one instruction per clock cycle. It comes with a dynamic branch prediction scheme and IEEE-754-2008 adherent floating point units, including fused multiply and accumulate units. The core is capable of supporting up to 512 GB of virtual address space using SV39 virtualisation scheme. The core supports the standard RV64GC ISA.

Core

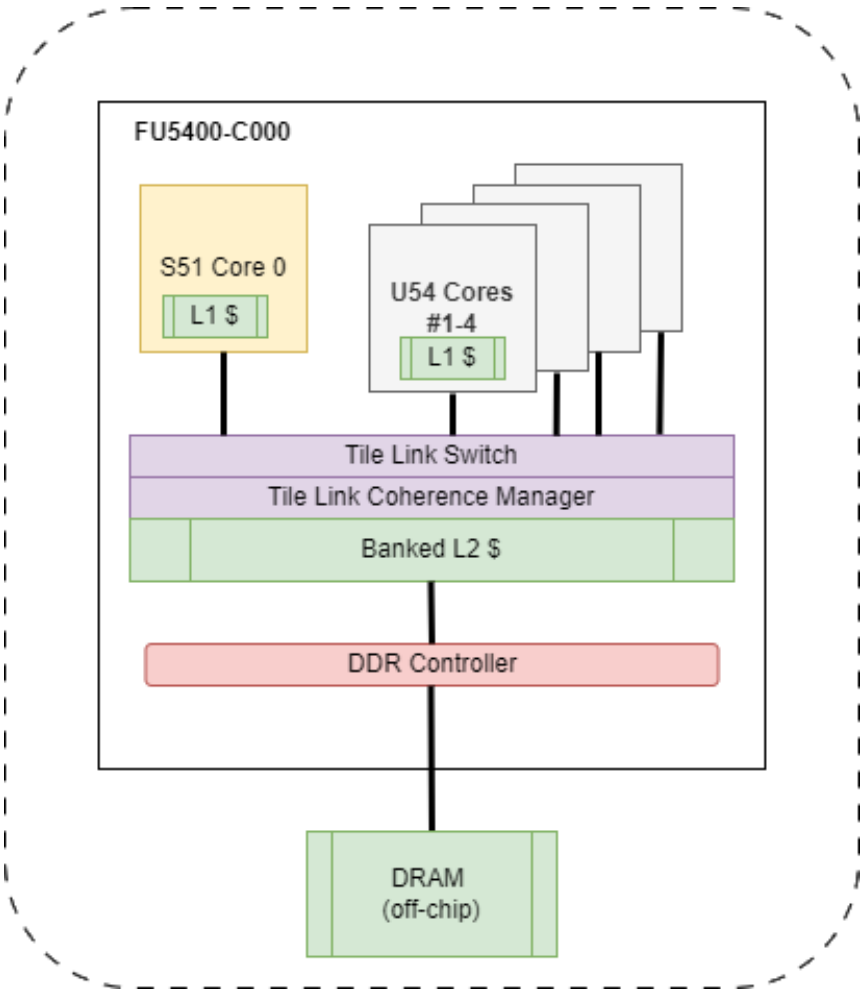


Figure 3.5: Sifive F5400-C000 Top Level Block Diagram

The U54 core is a single issue (scalar) 5-stage pipeline in-order core that supports Machine, Supervisor and User privileges modes. The U54 has support for Sv39 virtual memory support with a 39-bit virtual address space, 38-bit physical address space, and a 32-entry Translation Buffer. The U54 cores support data forwarding to overcome data dependencies. The pipeline comprises five stages: instruction fetch, instruction decode and register fetch, execute, data memory access, and register writeback. It has floating point units along with 30-entry branch target buffer (BTB) which caches the target of taken branches, a 256-entry branch history table (BHT), which stores the direction of conditional branches, and a 6-entry return-address stack (RAS). The latency of various functional operations are given below (refer Appendix B for details) :

1. LW has a two-cycle result latency, assuming a cache hit.
2. LH, LHU, LB, and LBU have a three-cycle result latency, assuming a cache hit.
3. CSR reads have a three-cycle result latency.

4. MUL, MULH, MULHU, and MULHSU have a 5-cycle result latency.
5. DIV, DIVU, REM, and REMU have between a 2-cycle and 65-cycle result latency, depending on the operand values.

The limited micro-architectural details of the U54 core poses a challenge for validating gXR5. The actual functional unit type and its latency, branch predictor used and its size, etc. need to modeled or existing models in gXR5 have to be tuned so as to reduce the disparity between simulator and actual hardware.

Memory Hierarchy

The freedom board comes with 32 KiB 8-way L1 Instruction and Data caches. The shared L2 cache is 2 MiB 16-way associative having coherency fabric. A DDR3/4 controller is used to interface the external DDR3/4/3L DRAM. The maximum data transfer rate from the DRAM used on board is 2400MT/s. The DDR subsystem operates on a separate clock running at 1/4 the DDR data rate.

The IP-protected U54 core with least micro-architectural details made public, provides a perfect avenue to test the soundness of the proposed validation methodology (Chapter 4). To prove the fidelity of the methodology, an open source hardware, namely Rocket core has also been chosen as the target hardware for validation of gXR5. The RISC-V based rocket system enjoys wide popularity among enthusiasts of open source hardware. The following section introduces the rocket system and its emulation on FPGA.

3.3.2. Rocket Chip

Rocket core is a 5-stage in-order scalar processor core developed at UC Berkeley and currently supported by SiFive. It implements the RV64G ISA and comes with Memory Management Unit (MMU) that supports page-based virtual memory and non-blocking data cache. Rocket also supports the RISC-V machine, supervisor, and user privilege levels.

The rocket core [13] along with L1 Cache is called a ‘tile’. The source code of the rocket tile is written in chisel (Constructing Hardware in a Scala Embedded Language [59]) that can be translated into verilog. It is high level abstraction language to describe hardware that makes configuring rocket with different components effortless. For example, the chisel libraries come with various types of adders implemented (ripple carrry, carry skip, etc.) that can be included in rocket core. Though this comes at the cost of less control over micro-architecture. Rocket makes use of UC Berkeley’s Chisel implementations of floating-point units [60].

The performance validation is done for a single rocket core emulated on Xilinx VC707 Virtex Evaluation Kit. The software stack used is U-boot, OpenSBI and the Linux 5.7 kernel. The following sections describe the hardware set-up and the rocket core architecture in detail. The optimized micro-architecture post synthesis on FPGA is discussed in Chapter 4. The discussion in this chapter has been restricted aspects of core that do not change with synthesis optimisation strategies.

Hardware set-up

The Chisel source code for Rocket Tile can be converted to synthesizable verilog targeting FPGA and ASIC [13]. The repository also provides source files to build C++ emulator and verilog files for RTL simulation. The VCS-synopsys tool [61] can be used to carry out RTL simulations with cross-compiled micro-benchmarks (available as part of 'tests' in RISC-V toolchain [62]). Open source Verilator [63] (verilog to C++) can also be used to simulate rocket core. However, the emulator provides insufficient details (such as cycle-accurate) on latency of different functional units. The various steps in translating chisel to synthesize-able verilog is given in Figure 3.6. The Chisel build dependencies include SBT and Mill. The Chisel source code is compiled to give intermediary FIRRTL files. The FIRRTL files are circuit level hardware description code that is compiled to emit verilog files for synthesizing on FPGA.

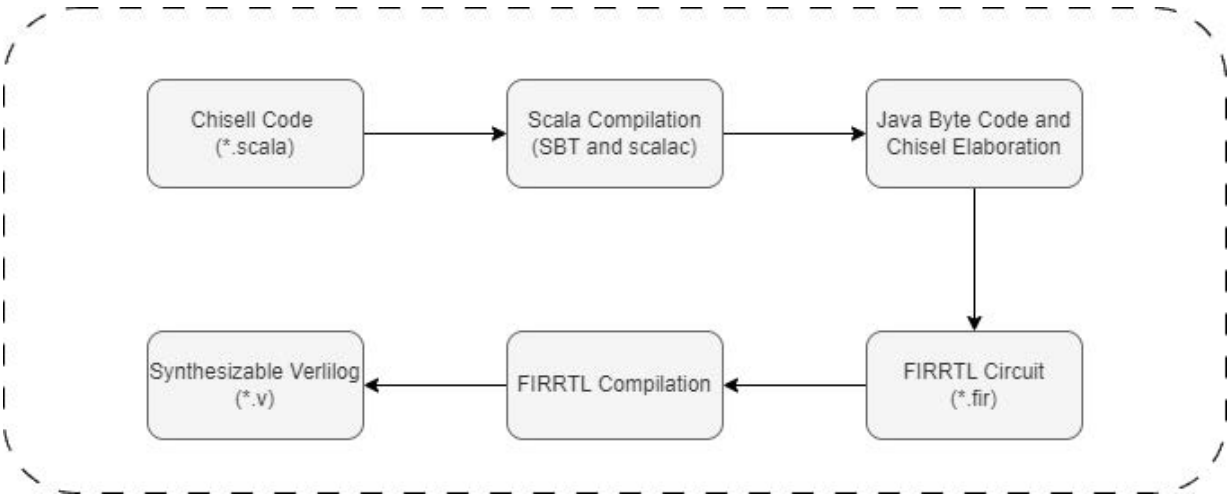


Figure 3.6: Chisel to synthesizable verilog translation

The rocket core can be configured to use two level cache heirarchy (for multi-core system) with coherency protocols. The emulated system uses one level cache (Instruction and Data) as depicted in figure 3.7. The caches are interfaced to the main memory via Tilelink. The DDR DRAM used is vivado's IP for DDR3 DRAM model. The UART is open source verilog [64] file used to interface with the system terminal. The rocket repository

provides package with definitions for generating JTAG bus interfaces. The repository also provides RTL packages using diplomacy to generate bus implementations of AMBA protocols, including AXI4, AHB-lite, and APB.

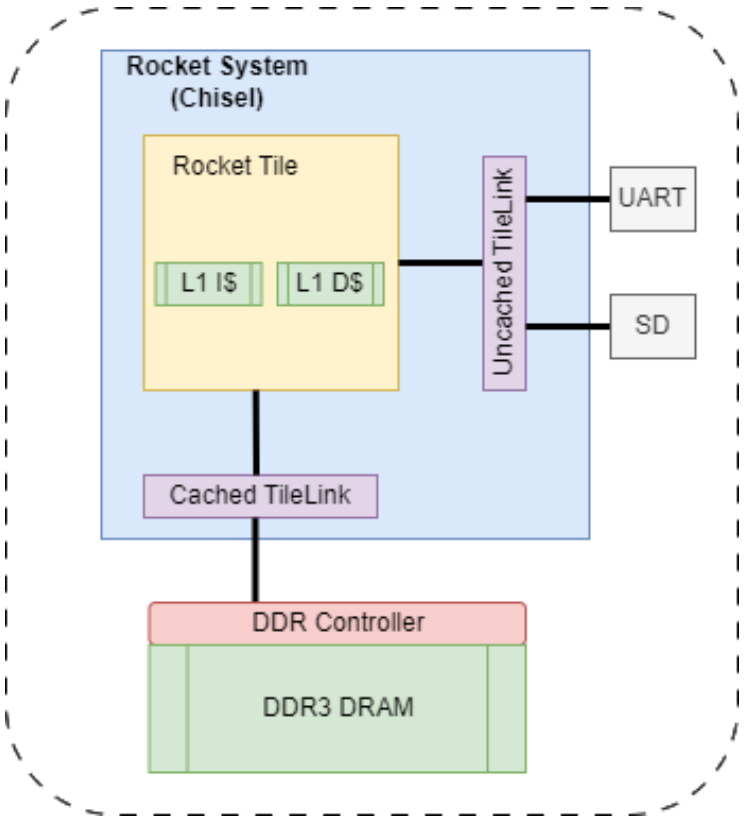


Figure 3.7: Rocket system Emulated on FPGA

Once the system is synthesised (using Vivado HLS [65]) and bitstream (and memory configuration file) uploaded on FPGA, a baremetal program is run on the emulated core using Xilinx System Debugger (XSDB) [66]. The Xilinx System Debugger uses Xilinx hw_server as the underlying debug engine. It can be used to put the cross-compiled elf file on to the HART (Hardware Thread). It provides the functionality of suspending the HARTs, repointing the Program Counter to the cross-compiled elf and executing the elf file. It also provides a mechanism to observe the contents of the register (including PC, SP, etc) change with the time. The bare metal program (.elf) helps in setting the jumpers, switches and matching the baudrate of terminal receiving the data from UART (FGPA). The bare metal program can be replaced by the actual bootloader, thereby confirming that the system has been successfully emulated. Further, QEMU can be used to debug and check the first stage bootloader by packing the firmware with next stage bootloader.

For running Linux on the emulated rocket system, the disk is created with two partitions, one for the bootloader and the other for rootfilesystem. The figure 3.8 represents the process of bootloading Linux. The I/O crossbar interfaces the Rocket system with

peripherals. The Default Program Counter points to the bootloader in the non-volatile memory (bootrom). The bootrom and the device tree are part of the bitstream uploaded on the FPGA. The program flow jumps to the First Stage Bootloader(FSBL) that copies the bootloader and VM Linux to the BRAM. The system then goes into User mode, relinquishing the control of program counter to the Operating System (Linux). The UART communicates to the host terminal and can be used to send command line instructions. Since the emulated system runs at a frequency of 100 Mhz and has low throughput, cross-complied benchmarks are put on the disk, instead of compiling them on the emulated system.

Following emulation of full-system on FPGA, the verilog source files along with chisel description of the hardware has been analysed to divulge the micro-architectural details of the emulated system. The system emulated is the default configuration of the rocket repository.

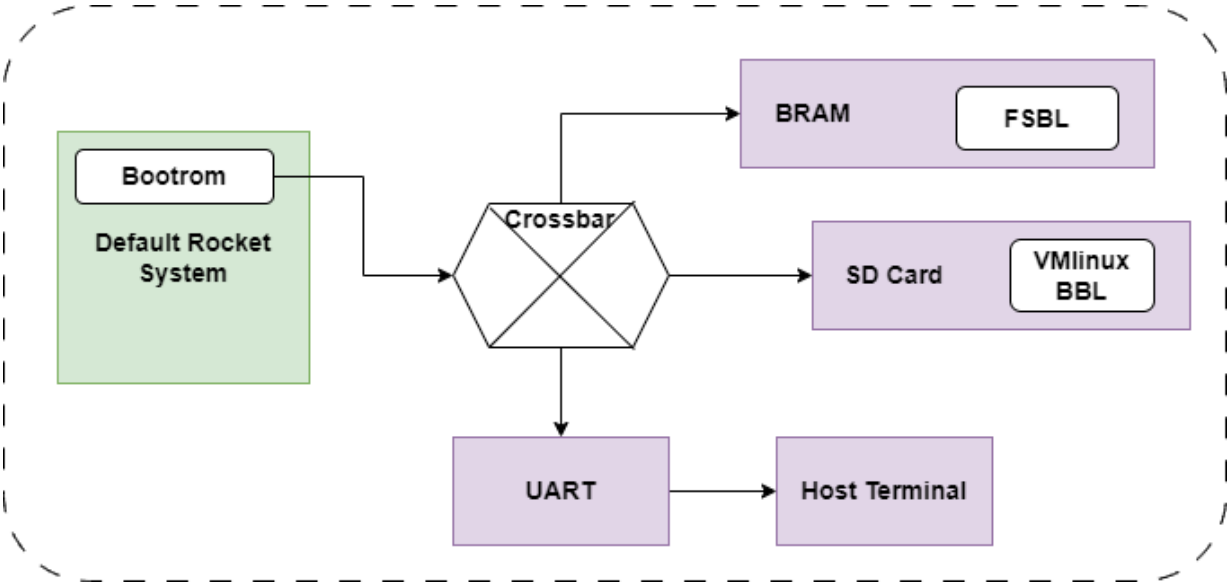


Figure 3.8: Booting Linux on rocket system emulated on VC707 FPGA

Micro-architecture

The rocket core comes in four flavors - ‘Big’, ‘medium’, ‘small’ and ‘tiny’. The features in each core has been summarised below:

1. **Big core:**
    - SV39 memory virtualisation support (Translation Buffers, Page Table Walkers etc.)
    - Floating Point Functional Units: Single precision and double precision fused

multiply and accumulate units with 3 and 4 clock cycles latency (as specified in chisel source code).

- Non blocking Instruction and Data caches.

2. **Medium core:** Identical to the Big core except for the lack of floating point units.
3. **Small core:** Identical to the Medium core except for the lack of memory virtualisation support.
4. **Tiny core:** Identical to the Small core except for the lack of Branch Target Buffer (BTB).

I base the validation effort against the Big core emulated on the FPGA.

Core

The Figure 3.9 shows the 5-stages of the rocket core. The first stage 'PC Generate' calculates the program counter for sequential fetches (pc+4) and branched fetches (BTB or target address calculation). The second stage 'Fetch'is associated with address translation and accessing the the Instruction from the cache. The third stage 'Decode' decodes the instruction into micro-operations. The fourth stage 'Execute' hosts the functional units followed by the last stage 'Memory' wherein the memory (or D cache) is updated . The 'Commit' stage can be clubbed with the 'PC Generate' state, making the pipeline 5-stages deep.

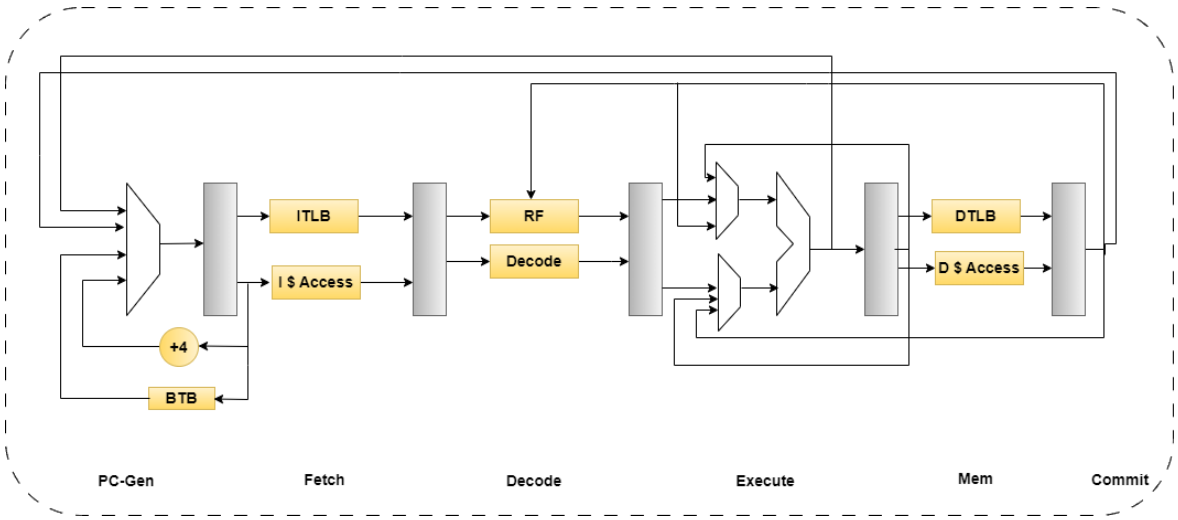


Figure 3.9: Rocket core: 5 stage pipeline [67]

The default configuration of Big core uses a Gshare branch predictor. The Gshare branch predictor is parameterizable and implements a hash of PC and global history register bits to access the Pattern History Table having 2-bit counters (strongly/weakely

taken/not-taken) for prediction. The equations 3.8 and 3.9 represent the hash function implemented in the Ghsare branch predictor.

$$Hash\_1(Global\_History) = Resize(Integer(\sqrt{\frac{3}{2}} * 2^{History\_length} * Global\_History)) \quad (3.8)$$

$$Hash\_2(Branch\_PC) = Resize(Branch\_PC >> log_2(fetch\ bytes)) \quad (3.9)$$

The resize function packs the computed output of the Hash match the length of number of entries of Pattern History Table. A high level view of the Gshare branch predictor is depicted in Figure 3.10. The branch predictor is complemented by 28 entry BTB table and a 6-entry RAS. The Big core comes with standard integer functional units. These

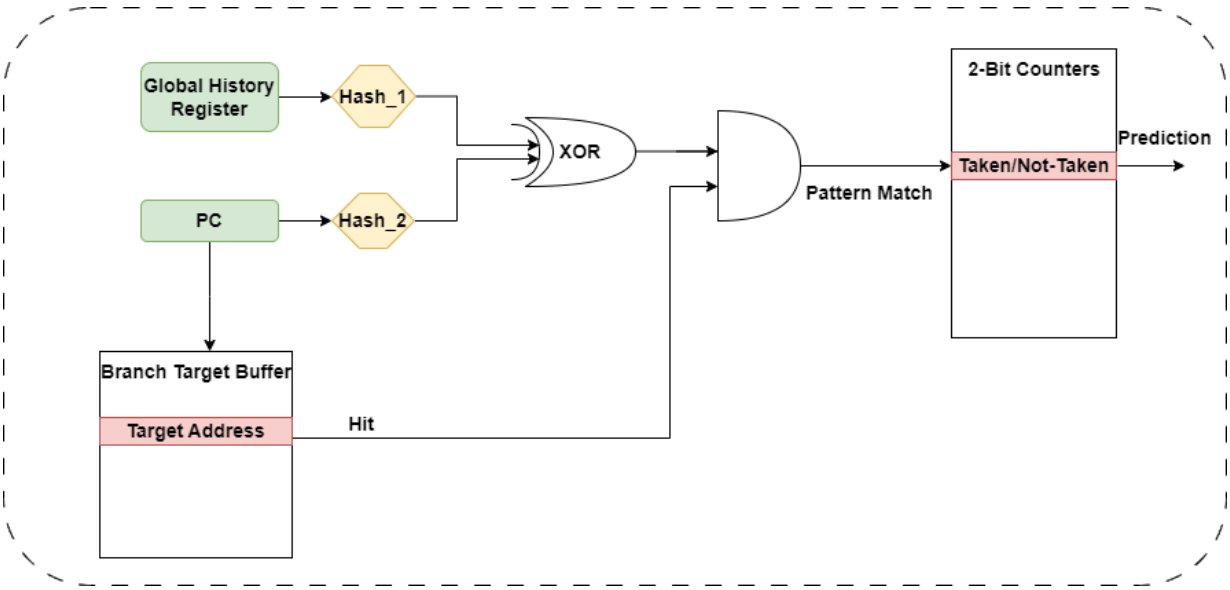


Figure 3.10: Rocket Core: Gshare Branch Predictor

include a 32-bit ripple carry adder (with one clock cycle latency) and an iterative multiplier that takes 8 clock cycles for 32 by 32 integer multiplication. It takes one clock cycle to load the multiplier with the operands and another clock cycle to put the result on the output bus. Hence, it has a total of 10 clock cycle latency. The latency of these integer functional units has been matched with the hardware.

Memory Hierarchy

The default configuration of the rocket tile was emulated. It has a 16KB 4-way associative L1-I and L1-D caches. The L1 Instruction cache is three stage pipeline with data access latency of 2 clock cycles under a hit. The L1 Instruction cache has been depicted in Figure 3.11. The virtual page number (VPN) to physical page number (PPN)

takes place in the stage 1 followed by tag comparison to check for corresponding data is cache. In case of a tag hit, the requested Instruction is sent to the CPU, otherwise the request goes down the memory hierarchy necessitating a cache block replacement.

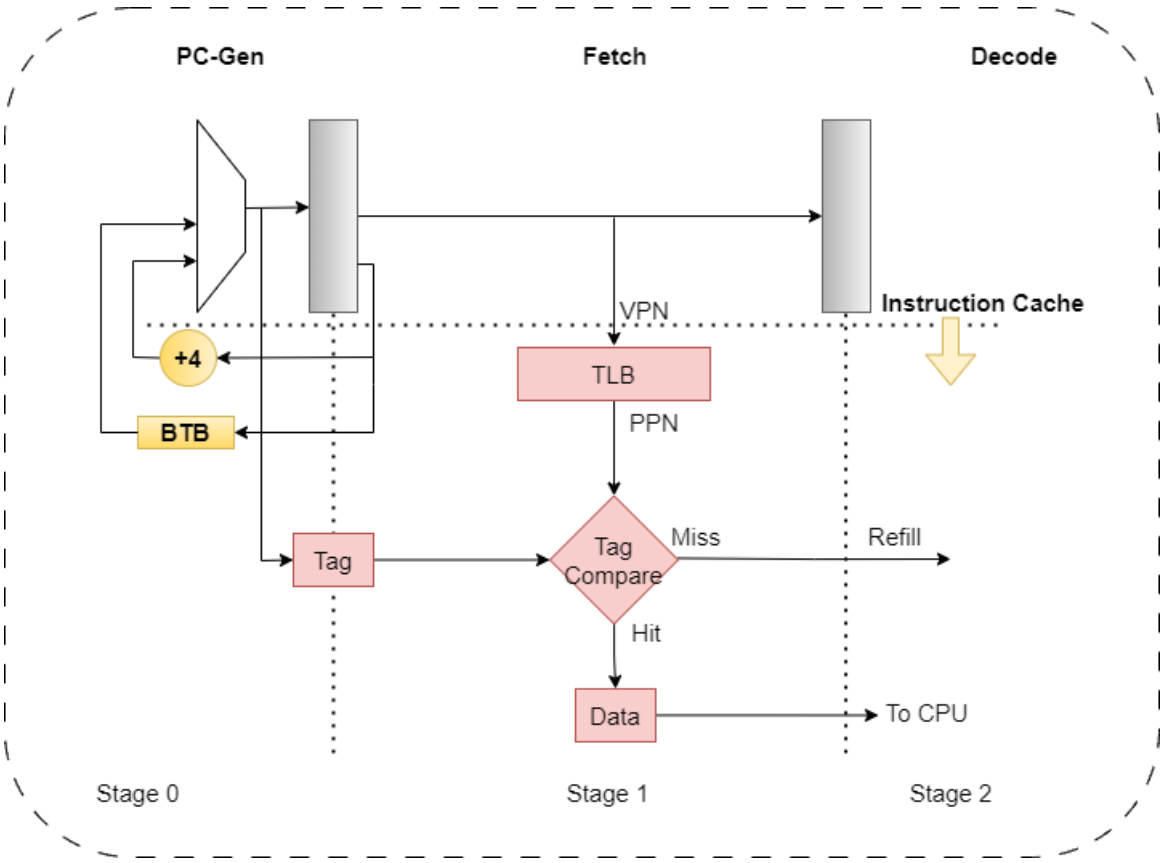


Figure 3.11: Rocket Core: L1-Instruction Cache

Similar to the instruction cache, the data cache completes the tag comparison and serves the request of data fetch in 2 clock cycles in case of a tag hit. There is an additional stage for hosting miss status holding register that keeps a track of hits under a miss. The L1 caches are multiplexed and interfaced directly to the main memory via Tilelink. A single core system emulated on FPGA does not need any cache-coherency protocol. However, the rocket core comes with the option of having shared L2 cache with coherency fabric supporting multi-core systems. The DDR3 RAM used is a single bank, 64 bit wide device giving a peak transfer rate of 1.6 Giga Bytes per second.

The gXR5 simulator uses in-order Minor CPU model for modeling both the Sifive and Rocket cores. DDR4_2400_16x4 and DDR3_2133_8x8 DRAM models are used for modeling Sifive board and Rocket system respectively. The frequency of CPU and DRAM models match the frequency of operation of the Sifive board. Whereas, for modeling the Rocket system emulated on the FPGA, modeling to actual frequency ratio of 10.665 is chosen for both CPU and DRAMs. This is due to the fact that frequency of operation of

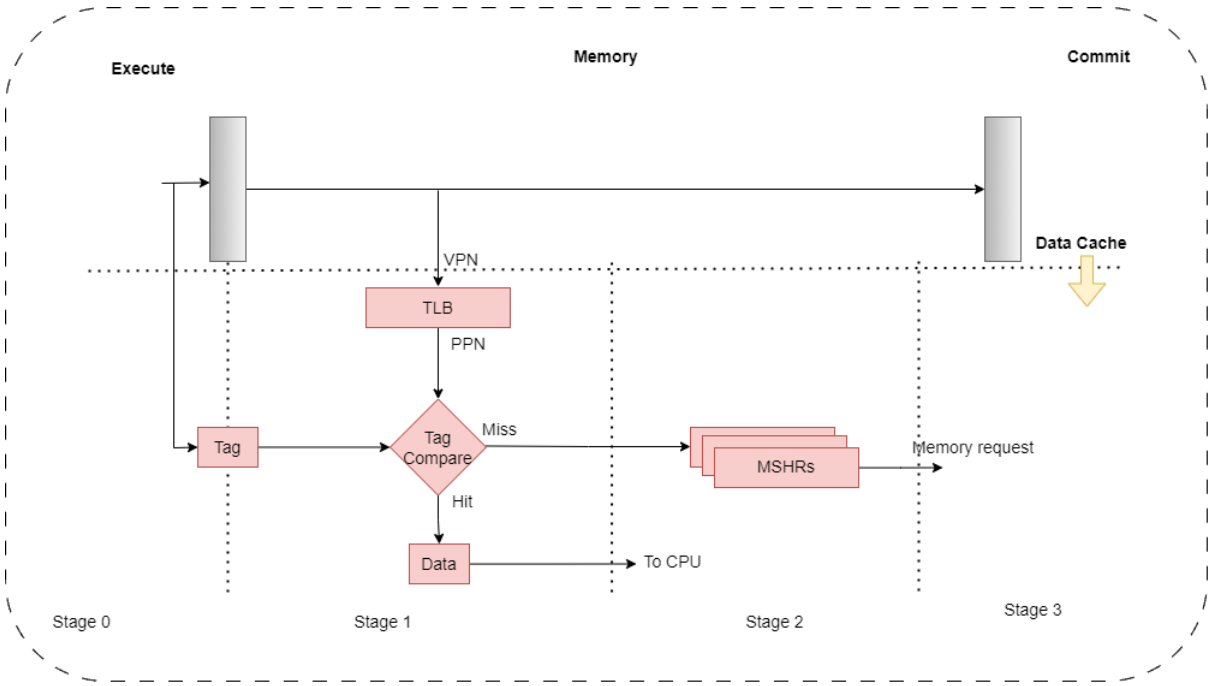


Figure 3.12: Rocket Core: L1-Data Cache

DRAM models in gXR5 can not be changed. The simulated and the hardware technical specifications have been summarised in the Table 3.1. The following Chapter introduces the performance validation methodology for gXR5. The methodology can be extended for other ISAs such as ARM, x86 etc. The chapter also gives insights into empirical analysis of Minor CPU model.

Component	HiFive Unleashed (Hardware Vs. Simulator)		Rocket (Hardware Vs. Simulator)	
CPU Core	U54	MinorCPU	Big Core	MinorCPU
CPU ISA Extension	RV64GC		RV64G	RV64GC
CPU Frequency	1 GHz		0.1 GHz	1.0665 GHz
L1 Instruction & Data Cache	32KB 8-Way		16KB 4-Way	
L2 shared Cache	2 MB 16-Way		None	
MMU	Sv39			
Modes	Machine, Supervisor, User			
RAM	DDR4	DDR4_4x16	DDR3	DDR3_8x8
RAM Frequency	2400 MHz		200 MHz	2133 MHz
RAM Size	8GB		4GB	
System Bus	TileLink	XBar	TileLink	XBar

Table 3.1: Technical specifications of simulated models and target hardware.

4

Methodology

"The mere act of observing the system
changes the state of the system"
-Observer Effect

The Chapter introduces a new methodology of calibrating the MinorCPU model in gXR5. The methodology deviates from the conventional methodology of "micro-architectural" level calibration using Hardware Performance Counters as discussed in Chapter 2. The proposed methodology uses component level calibration, proving to be much faster and arguably more accurate than the existing methodology. CPU model, namely, MinorCPU is calibrated against Sifive Highfive Unleashed Freedom U54 core as well as against "Big-core" of the Rocket-chip emulated on VC707 FPGA. The Chapter begins with profiling of selected benchmark suites used for calibration and delves into the empirical analysis of the performance statistics of the simulated hardware. This is followed by fine-tuning of the models to reduce simulated performance discrepancy compared to the target hardware. This chapter contains the majority of the thesis contributions.

4.1. Benchmarks: stress-ng

The stress-ng benchmarks were originally designed to perform accelerated stress-tests of a particular component of the computing system and to cause thermal overruns. The benchmark suite has been divided into classes of "stressors", with each class stressing a particular component. For example, the memory class of stressors stress the main

memory of the system. The stressor classes are further divided into methods. The stress-ng benchmarks are a collection of micro-benchmarks. They are more representative of time complexity of actual workload than micro-benchmarks and at the same time they give enough granularity at the level of micro-architecture so as to calibrate the simulator against the hardware. This proposed methodology of using stress-ng benchmarks is called “component-level” calibration as against “micro-architectural-level” calibration. The Figure 4.1 captures this idea. The stress-ng CPU class benchmarks can be further classified into Control, Memory and Compute/Arithmetic intensive. The section 4.1.2 discusses the metrics used to classify the stressors into above mentioned categories.

The stress-ng benchmarks measure the performance of the underlying hardware through a metric called ”bogo ops” (or rate, bogo ops/second). Bogo ops are representative of the workload and gives a rough estimate of the performance. One bogo op is one loop iteration of stressor action. For example, ”sqrt” stressor method of class CPU finds the square root of a random number. Finding square root of one random number amount to one bogo op. This benchmark when run on hardware for 5 seconds reports 2388 bogo ops. Hence, square roots of 2388 random numbers are calculated at a rate of 477.6 (bogo ops/s) per second. Another example is of the CPU class stressor with method ”prime” that finds the first 10000 prime numbers using a slightly optimised brute force naïve trial division search. The selected benchmark suite comprises of the following ten methods of class CPU: rand48, prime, queens, stats, trig, int64longdouble, longdouble, intconveresion, matrixprod, and sqrt. The detail functionality of these microbenchmarks is in the Appendix D.

Before starting to validate the simulator, it is necessary that the workload (in this case stress-ng benchmarks) are using the simulator. This gives the much needed details for the designer to understand the (micro)architecture of the simulated models.

4.1.1. Profiling stress-ng Benchmarks

The selected CPU class stress-ng benchmarks were profiled using gXR5 to understand the workload characteristics. The source code of the stressors were analysed to get insights such as working set size and control complexity (Table 4.3). As mentioned in Chapter 1, Figure 1.1 the gem5 decoder maps the instructions to ‘op-classes’. These op-classes are then assigned to a functional unit inside gXR5. gXR5 simulations statistics report the relative number of the instructions executed belonging to each op-class/functional unit. The intensity of use of various functional units varies for each benchmark. This has been depicted in Figure 4.2. All the selected stressors have high Integer Functional Units

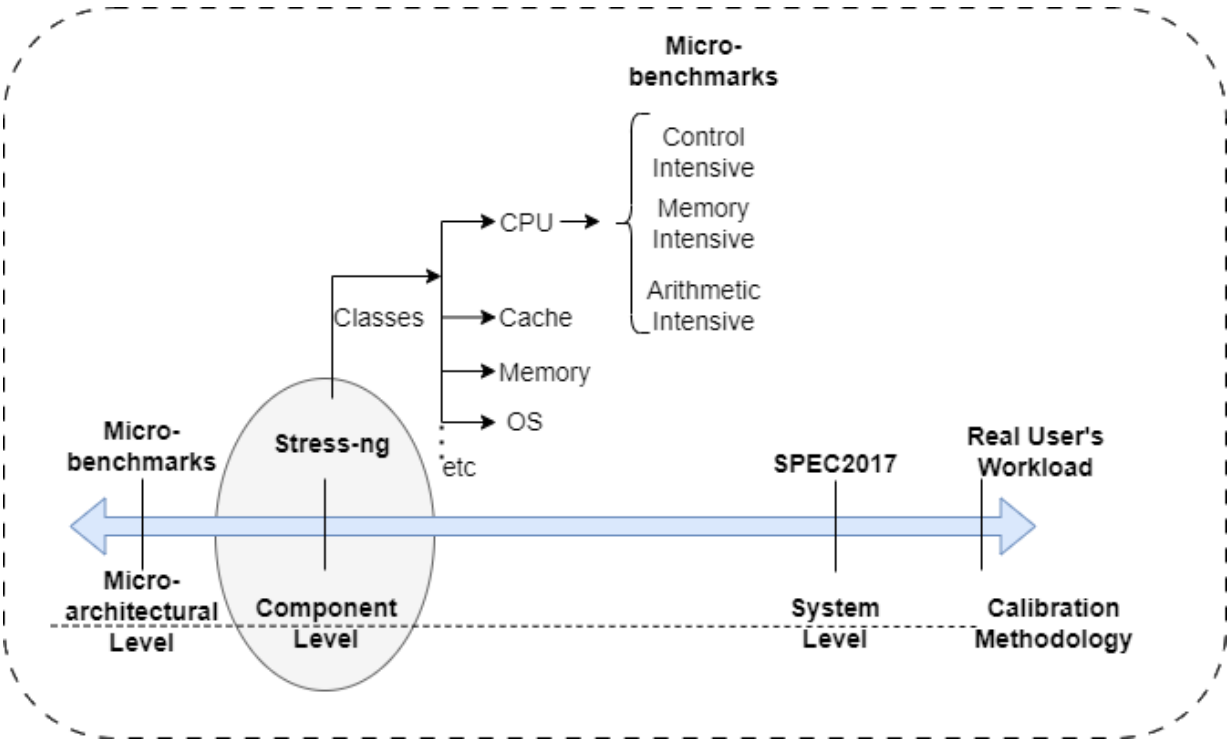


Figure 4.1: Methodology: Component Level Calibration

utilisation (60 to 99%). The Integer Functional Units include IntALU, IntDiv and IntMult functional units each having the corresponding op-class. Similarly, the Floating Point Units have multiple functional units as depicted in the Table 4.1.

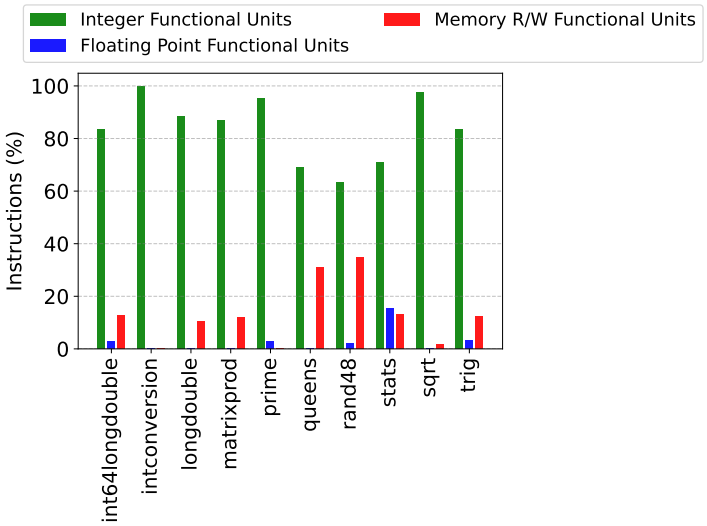
The stressor ‘stats’ has considerable floating point operations whereas the ‘queens’ and ‘rand48’ have significant Memory R/W (Load/Store) operations. The Working set size further decides whether the Memory operations would be caches or main memory access. Analysing the source code of the stressors, the size of the data being operated upon can be assessed. This ‘working set size’ of the stressors is summarised in the Table 4.3. The working set along with the size, associativity and clusivity of caches decides the exact cache/memory level being stressed by the stressor.

IntFU			FloatFU							
IntALU	IntDiv	IntMult	FloatAdd	FloatCMP	FLoatCvt	FloatMult	FloatDiv	FloatMisc	FloatMultAcc	FloatSqrt

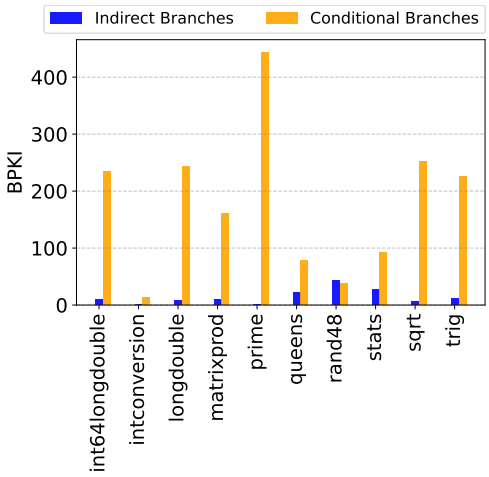
Table 4.1: Instruction Operation Classification by Utilized functional unit

Memory Read/Write Functional Units			
MemRead	MemWrite	FloatMemRead	FloatMemWrite

Table 4.2: Instruction Operation Classification by Utilized Memory R/W functional units



(a) Functional Unit Utilisation



(b) Conditional and Indirect Branch Instructions

Figure 4.2: Profiling stress-ng benchmarks for (a) functional unit utilisation and (b) branch incidence (using Branches Per Kilo Instruction (BPKI) as the figure of merit).

The benchmarks' branch characteristics were also profiled. A higher branch incidence in a benchmark sets it apart from others and has to be tuned separately. The Figure of Merit chosen to profile the benchmarks was "Branch per Kilo Instructions" (BPKI). Figure 4.2 depicts the incidence of Branches for the stressors. The control structure of stressors has been summarised in the Table 4.3. Most of the stressors have a regular for loop. The 'queens' and 'matrixprod' have nested "for-while-if" statements and nested "for-for-for" loops. The 'sqrt' stressor has "if-else" inside regular "for" loop.

Once the benchmarks were profiled on Baseline gXR5 configuration (refer Appendix **??**), the classification of benchmarks was necessary to streamline the calibration effort for reducing disparity between gXR5 and the target hardware. The classification helps in ensuring that rest of the stressors are not 'de-tuned' while tuning for a particular stressor. The following section discusses the metrics chosen to further classify the CPU class stressors.

4.1.2. Classification of Benchmarks

The initial profiling carried out is used to classify the CPU stressors into three categories:

- Arithmetic/Compute Intensive: Having 20 % or more integer/floating point 'op-classes' or functional unit utilisation.
- Memory (Load/Store Intensive): Having 20 % or more read/write memory functional

Benchmarks	Working Set Size (Bytes)	Control-Structure
Int64LongDouble	80	for
IntConversion	32	for
Queens	64	for-while-if
Stats	2096	for-if
Trig	36	for
Prime	16	for-if
Matrixprod	196,624	for-for-for
LongDouble	32	for
Rand48	20	for
Sqrt	84	for-if

Table 4.3: Benchmarks Analysis

unit utilisation.

- Control Intensive: Having at least 100 conditional BKPI incidence with nested control statements.

This classification of benchmarks facilitates tuning for complementary and mutually exclusive stressors. For example, stressors that are compute/arithmetic intensive can be calibrated for just by configuring Integer/Floating Point functional units. At the same time, tuning for control intensive stressors would not lead to de-tuning of arithmetic intensive stressors. This leads to stressors falling into (at least) one of the three categories as summarised in the Table 4.4. All the stressors are arithmetic intensive. Two stressors, namely ‘Queens’ and ‘Rand48’ are memory (Load/Store intensive). The Load/store instructions might be accessing the caches, RAM, or storage. With the exception of three stressors, ‘Rand48’, ‘Intconversion’ and ‘stats’ all the stressors are control intensive. The Classification of benchmarks aligns with similar work done for x86 [51] and ARM cores [54].

Benchmarks	Arithmetic Intensive	Memory Intensive	Control Intensive
Int64LongDouble	✓		✓
IntConversion	✓		
Queens	✓	✓	✓
Stats	✓		
Trig	✓		✓
Prime	✓		✓
Matrixprod	✓		✓
LongDouble	✓		✓
Rand48	✓	✓	
Sqrt	✓		✓

Table 4.4: Classification of Benchmarks

4.2. Validating against Sifive Unleashed

The stressors are run on Sifive board for an execution time of 5 seconds. The only Hardware Performance Counters used were the instructions and the clock cycles. The hardware characterisation is significantly reduced owing to the methodology of targeting IPC of hardware and the simulator. The following section applies the proposed methodology to calibrate for the Sifive U54 core. The same set of stressors were executed in simulator for 5 seconds.The U54 core supports hardware performance monitoring facility compliant with the RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10.

4.2.1. MinorCPU

The MinorCPU model, as discussed in Chapter 3.2 gives fine granularity in modelling the micro-architecture of an actual CPU. Some of the important tunable attributes of the Minor CPU model are summarised below:

- *Fetch1toFetch2Delay*: Delay in clock cycles from fetch1 to fetch2 stage.
- Size of input buffers to fetch2, decode and execute stages
- Number of simultaneous accesses to memory allowed.
- fetch2 to fetch1 (backward) delay in communicating the branch prediction decision.
- execute to fetch1 delay for communicating the branch execution decision.
- Size of the Load and Store queues.
- Number and Latency of the functional units.

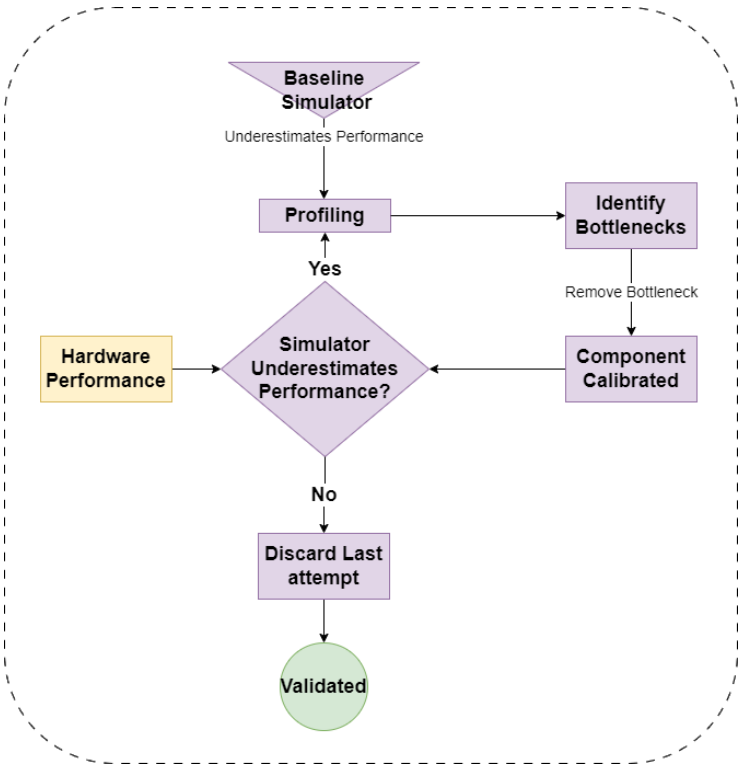


Figure 4.3: Calibration strategy

A large number of attributes (such as listed above) helps in bringing the simulated models close to the actual hardware and have high level of configurability (for modeling different hardwares). However, the down side of having large number of attributes is the huge design space that needs to be explored. Having a strategy to implement the methodology was instrumental to reduce the design space. The baseline simulator underestimated the performance of almost all the stressors compared to the actual hardware. The Figure 4.3 captures this strategy. The baseline simulator's functional unit latencies in MinorCPU model was matched with the latency of the functional units of U54 core as described in section 3.3.1. This is called the 'baseline' simulator. The problem at hand becomes a challenge of identifying bottlenecks in the simulator untill its performance is at par with the actual hardware. The Figure 4.4 depicts the performance of the baseline simulator compared with the hardware for the selected CPU stressors. Performance of all, except one ('Stats'), stressor is underestimated by the simulator. This is because the baseline simulator has multiple floating point op-classes mapped to the same functional unit. Since 'stats' is the only application that has considerable floating point operations, the simulator over-estimates the performance for 'stats'.

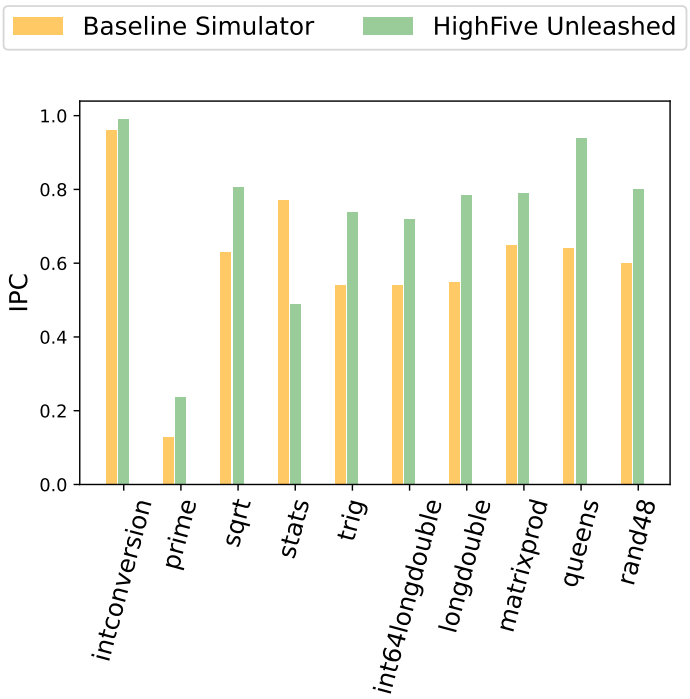


Figure 4.4: Comparison of Baseline simulator and hardware running stress-ng benchmarks

Calibrating Arithmetic Functional Units

The baseline simulator has a pool of functional units instantiated. These are integer and floating point multiply, divide, Square-root, fused multiply and accumulate and integer functional units. Each functional unit has tunable attributes such as latency (of operation), Source Register Relative Latency, Extra Assumed Latency, etc. The latency of operation is the number of cycles the functional unit takes to execute the micro-operation. The Source Register Relative Latency models the number of clock cycles the data required by particular functional unit is available in the source registers. Expiry of the said clock cycles leads to new data being put from memory/Cache. Increasing this value can lead to memory read operations taking longer (i.e., stalls) because of unavailability of register files, whereas, reducing the latency puts pressure of L1-Data cache. Extra Assumed Latency has been introduced to give the designer another variable to fine tune the functional unit operational latency.

A preliminary (functional unit) design space exploration is performed to identify plausible sources of performance bottleneck (in CPU). The latency of the various functional units are decreased to find the effect on the performance. The decrease in latency of most functional units leads to increase in stall cycles thereby implying that the functional units such as integer multiply/ALU and floating point functional units are not the bottleneck.

However, the integer division unit greatly affects the performance of the stressor ‘prime’ as it has more than 20 % of the total Instructions using division functional unit. The latency of integer division unit is explicitly specified to be between 2 and 65 clock cycles. The ‘prime’ stressor was used to fix the latency of the Integer division functional unit. A latency of 19 clock cycles for the integer division unit brings the performance statistics for the ‘prime’ stressor (having high division functional unit utilisation) closer to actual hardware performance results. The IPC error for Prime benchmark reduces from -77% to -2% (negative values denote that the simulator underestimates the IPC).

The integer and floating point functional units are not the bottleneck for performance. A decrease in latency leads to the CPU idling which indicates that its functional units are awaiting new data or instructions to be operated upon. Hence, the bottleneck is either memory bandwidth or control hazards. Data dependency is ruled out as the MinorCPU model supports data forwarding. The Data path to memory travels via the memory read and write functional units. Hence, the attributes of these units were calibrated before calibrating the L1 level caches. These Memory Read/Write functional units model the front end of memory latency.

Calibrating Memory Execution Units

The memory read and write functional units are the next most utilised functional units. Calibrating these functional units is expected to reduce the IPC error of the memory intensive benchmarks. Since these affect the front end latency of the memory, reducing the read/write functional unit latency will reduce the memory access latency related stalls in the execute stage.

The operational latency of these units is reduced to 2 and 1 clock cycles, respectively. A further decrease in latency leads to overestimation of IPC for Memory Intensive benchmarks. Moreover, reducing the Source Register Relative Latency adversely affects the performance as it leads to more L1 data cache accesses and thrashing. This is due to the fact that operand in source registers is short lived and has to be fetched again from the data cache when micro-ops sit in the issue buffer awaiting functional units to be available. The MAPE in IPC for the selected stressors reduce from 36% to 19.9%.

The MinorCPU model exhibits ‘decoupling’ of CPU stall cycles and IPC. Conventional wisdom suggests that an increase in CPU stalls will lead to reduction in IPC. In case of control hazards, the CPU stalls (Fetch2 and Fetch 1 stages) but the presence of buffers in the Decode and Execute stages ensures that the functional units are busy executing the instructions prior to branching. For example, in a large input buffer to Execute and Decode stages can lead to higher IPC even though there are increased

CPU stalls due to branch misprediction. A sample RISC-V assembly code is depicted in Figure 4.6 along with the corresponding micro-operations utilizing various functional Units. The pipeline timing diagram has been drawn (Figure 4.5) to highlight the role of buffers between stages. The fetch width is taken as 2 instructions wide, with at-most one instruction coming out of the pipeline every cycle (scalar).The highlighted (orange) slots depict the status of processor with instructions sitting in issue buffer of the Execute unit. For the sake of simplicity, buffers to other pipeline stages are kept as one instruction wide. E1, E2 represent memory read and integer(ALU) functional units respectively. Since there is only one functional unit of each type, there exists dependency because of the resource availability (functional units). However, E1 and E2 can be used in the same clock cycle by two different instructions (e.g. at clock cycle number 14, instructions i2 and i8 utilize different functional units).It is assumed that the conditional branch i5 has been mispredicted (taken) whereas i8 has been rightly predicted (taken) by Fetch2 unit. Branch misprediction leads to flushing of instructions i6,i7 and i8 (highlighted in red). The instruction i9 is flushed because of branching at instruction i8. Even though the branch misprediction causes pipeline flushing (and associated penalty), the issue buffer keeps the functional units busy. It is only at clock cycles 9 and 11 that no instruction comes out of the pipeline. The timing diagram is analytical and is used to provide an explanation as to the empirical observation of stall cycles-IPC de-hyphenation. This could be confirmed from the C-code of the pipeline (refer Section 3.2), wherein the stalls are propagated backwards (from Execute to Fetch1 stage) as the method ‘Evaluate ()’ is applied on Execute stage first and subsequently to previous pipeline stages. Hence, there exists a scenario where the Execute stage does not stall (and has high functional unit utilisation) whereas the fetch1 stage stalls.

Calibrating the functional units reduces the MAPE in IPC for selected stress-ng benchmarks from 36% to 19.9%. Once the memory functional units are calibrated, the focus is on mitigating the control hazards. The Fetch2 unit hosts the Branch predictor unit and hence, the tunable attributes of the Fetch2 unit are taken up for design space exploration and understanding their significance in reducing control related bubbles in the pipeline.

Calibrating the Branch Execution Unit

Control hazards adversely affect the performance of an in-order single issue core. The baseline configuration of MinorCPU model uses a Tournament Branch Predictor with a 32-entry Branch Target Buffer (BTB), 256-entry Branch History Table (BHT), and 6-entry Return Address Stack (RAS). The size of BTB, BHT and RAS conform to the actual hardware specifications. The specification sheet of U54 core does not unveil more about

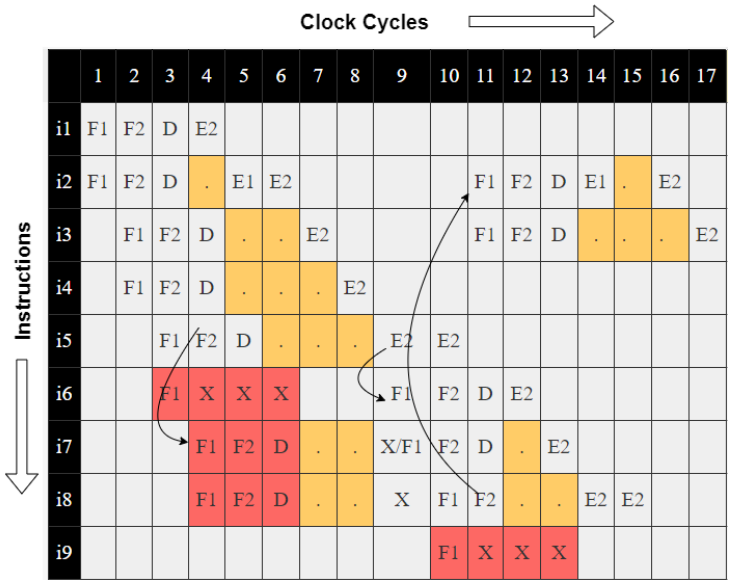


Figure 4.5: An example of control related stalls

the type and the micro-architecture of the branch predictor used.

In order to study the effect of branches on performance for the entire set of benchmarks, the baseline configuration is profiled with a focus on branch predictions. The profiling is carried out for conditional branches and indirect branches. Conditional and indirect branches make up the majority of branches across all benchmarks as depicted in Figure 4.2. The incidence of unconditional branches (function calls) are insignificant and thus ignored. The Fetch2 to Fetch1 backward delay can reduce the penalty associated with branch mispredictions. This is empirically verified as the delay between Fetch2 stage and Fetch1 stage affects the IPC of almost all the benchmarks. A delay of zero clock cycles implies that the branch prediction happens in the same clock cycle as instruction fetch. Since the forward delay between Fetch1 and Fetch2 stages is one clock cycle, setting the backward delay to zero ensures that a correct branch prediction has a latency of one clock cycle. This conforms to the specification of actual hardware. Wang et al. [54] observed a similar improvement in IPC error for control intensive benchmarks run on Out-of-order CPU model in gem5 by calibrating the backward delay. Moreover, the attribute Branch Execute Delay models the latency between Execute unit directing Fetch1 unit to fetch new stream of instructions. The branch delay latency enables the branch slots to be occupied for unconditional branches. However, it is expected to adversely affect the performance in case of conditional and indirect branches. The baseline value of 3 clock cycles branch delay slot minimises the error in IPC for control intensive benchmarks.

The above discussed parameters affect the efficiency of pipeline flushing in case of branch misprediction. Since the input buffers to Decode and Execute stages keep the

```
start:                  Instruction_Number  Mirco-ops using FUs
   ADD  t0, zero, zero          1           1 -> ALU (Add operands)
   next:
   LD t1, 102                   2           2 -> ALU (add offset), mem read FU
   ANDI t1, a1, 1               3           1 -> ALU (Logical AND)
   SRAI a1, a1, 1               4           1 -> ALU (Shift Right)
   BEQ  t1, zero, skip          5           2- > ALU(comparison),ALU(addr calc.)
   ADD  t0, t0, a0              6           1 -> ALU (Add operands)
skip:
   SLLI a0, a0, 1               7           1 -> ALU (shift right)
   BNE  a1, zero, next          8           2 -> ALU(comparison),ALU(addr calc.)
   ADD  a0, zero, t0            9           1 -> ALU (add operands)
   HLT                          10          1 -> NOP
```

Figure 4.6: Sample RISC-V Assembly Code

functional units occupied, an appropriate backward latency of Execution to Fetch1 unit can ensure that the bubbles due to control hazards in pipeline do not have cascading effect. At this stage, the difference in IPC of simulated and hardware performance

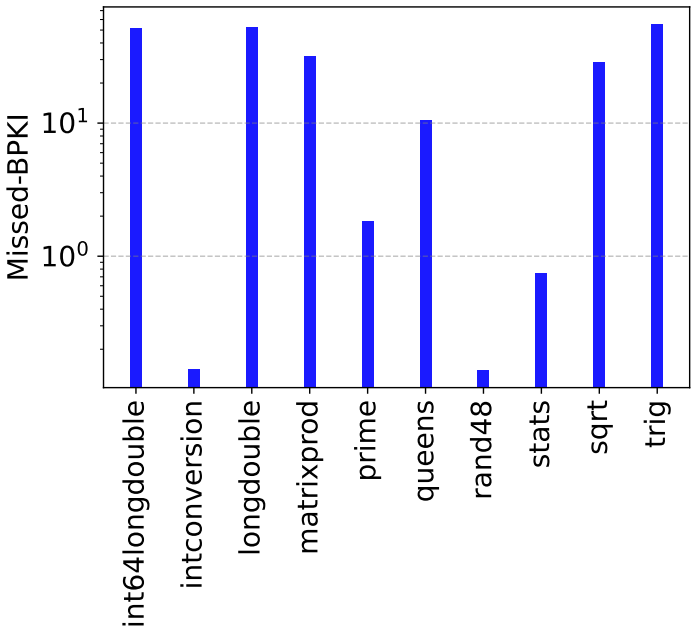


Figure 4.7: Missed-BPKI for default tournament branch predictor

statistics can be attributed to the fact that the hardware micro-architecture of the branch predictors deviates a lot from the gem5 model of branch predictors. Hence, a design space exploration is performed to bridge the gap between simulated and actual branch predictors.

Design Space Exploration of Branch Predictors

Though control hazards can be mitigated by adopting fine grained multi-threading, predicated execution, multi-path execution etc, the most popular technique for mitigating the stalls related to control hazards is by performing branch predictions. In branch prediction, the program counter is speculatively decided to fetch new set of instructions. Hence, the new set of instructions can be fed to the pipeline without waiting (bubbles in pipeline) for the outcome of the previous instructions. The downside is that the branch mis-predictions necessitate flushing of the pipeline to get rid of wrongly fetched instructions. Section 3.2 provides a brief overview of the various branch predictor models available in gXR5. These models come with squashing logic to take care of the branch mis-predictions and restoring the state of the processors.

The focus of study is conditional branch predictions. Figure 4.7 depicts the Mispredicted Branches per Kilo Instructions (Missed-BPKI) for various benchmarks. A high branch misprediction rate also leads to thrashing in the BTB and hence a poor BTB hit rate for many of the benchmarks. The underestimation of IPC by the simulator can be attributed to the poor performance of branch predictors [54].

The gXR5 has five branch predictors: Tournament, Local, BiMode, Multiperspective and TAGE branch predictors with tun-able size of Branch Table History and Branch Target Buffer. The misprediction rate of various Branch Predictors is shown in Figure 4.8.The baseline tournament branch predictor has the highest variance in misprediction rates. The Local and Bimode branch predictors perform slightly better than the Tournament branch predictor in terms of prediction rate, yet the median misprediction rate is still very high for most of the control intensive benchmarks. The Multiperspective Perceptron (MP) branch predictor [56] and TAGE branch predictor have lower median branch prediction miss rate. The TAGE branch predictor has larger outliers that increases the Mean Absolute Percentage Error in IPC compared to MP branch predictor. Though the median miss prediction rate is 10% for MP branch predictor, it is still very high given the advancements branch prediction has witnessed over the last few decades. Implementing the Multiperspective Perceptron branch predictor along with tuning the branch execution unit reduces **the Mean Absolute Percentage Error (MAPE) in IPC from 19.9% to 11.8 % for the entire benchmark suite. The MAPE in IPC for control intensive benchmarks reduces from 15% to 9%.**

A high branch misprediction rate not only increases thrashing in the L1-Instruction cache by fetching wrong set of instructions speculatively, it also causes high L1-D cache miss rate by removing a relevant block of data from L1-D cache. This necessitates to calibrate the L1 level caches at the very end of calibration effort.

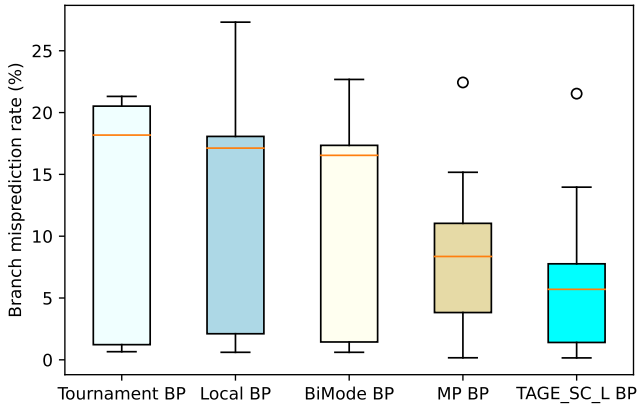


Figure 4.8: Misprediction rate for different Branch Predictor

4.2.2. Caches

The baseline cache policy is exclusive implying that the data block thrown out from L1-D cache has to be fetched from Memory (instead of L2 cache). The figure of merit chosen to profile the simulator is the Average Memory Access Time (AMAT). The design space for reducing the AMAT has been described in Section 3.2. The clusivity policy for L2 cache is changed to 'mostly inclusive' to reduce the AMAT in case of L1 cache misses. As expected, the AMAT for stressors whose working set resides in L2 cache reduces the most. This has been depicted in Figure 4.9. Moreover, the data access latency and response latency of L1 cache has been reduced to further reduce AMAT for both Data and Instructions. (refer appendix **??**). The calibration effort has been restricted to L1 caches as the CPU class stressors do not stress the memory hierarchy.

The simulator models were configured to match the hardware specifications available publicly. This baseline simulator had an error of 26 % that was brought down to 14 % for selected stress-ng stressors. The biggest contribution in reducing the performance disparity was the branch predictor. Just by changing the branch predictor model to Multiperspective Perceptron (64KB) leads to 8% reduction in error. Though the focus was on calibrating the MinorCPU model to reduce IPC error, L1 caches were also calibrated as the stressors have significant L1 cache accessses (Load/Store instructions with working set size less than L1 cache size).

The Figure 4.10 depicts the IPC of simulator at different stages of calibration. The IPC for most of the stressors increases at every calibration stage, bringing the performance of the simulator closer to the actual hardware. Hence, overall error in the simulator reduced

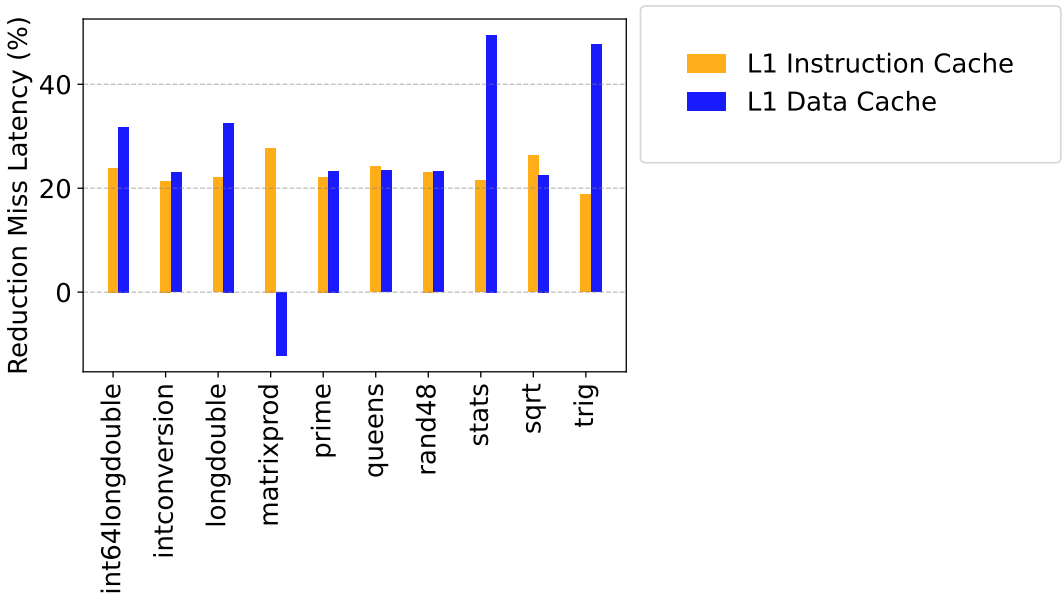


Figure 4.9: Reduction in average miss latency for L1 Instruction and Data caches

considerably for the stress-ng benchmarks. The stress-ng micro-benchmarks help in calibrating the modeled simulator rather quickly. A single run of the stressor takes 3 hours in the simulator and 5 seconds in the hardware. Compared to running the macro benchmarks (which have simulation time of 1 to 7 days), considerable simulation time has been saved. However, the simulator needs to be tested with real-user's workload. Chapter 5 discusses the performance validation of simulator with SPEC2017 suite as the 'test' set.

The proposed methodology is now applied to an open source RISC-V system (Rocket). The following section calibrates the MinorCPU model against the Rocket core using the same selected stress-ng CPU class stressors. The objective is to prove the fidelity of the proposed methodology by applying it to a different target hardware.

4.3. Validating against Rocket System

The default configuration of the rocket core comes with floating point functional units, namely, single precision fused multiply and accumulate (SPFMA), double precision fused multiply and accumulate (DPFMA) and division unit. The tournament branch predictor has been used in the simulator as there exists no Gshare branch predictor model in gXR5. The latency of integer functional units has been matched with that of the rocket core. The L1 level cache size, associativity and data access latency has also configured

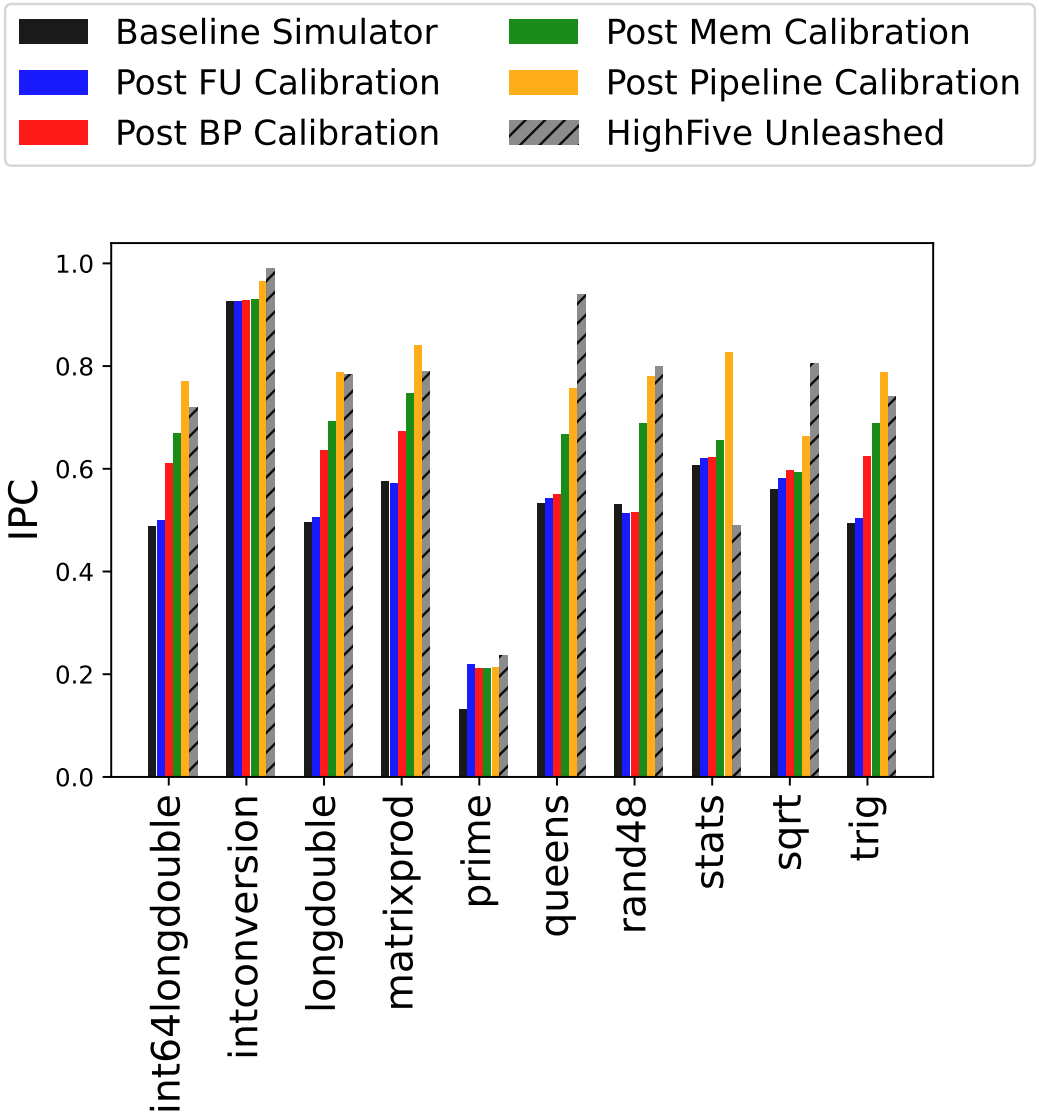


Figure 4.10: Simulator Vs. Hardware IPC at different calibration stages

exactly as that of the rocket system. This configuration is called the 'Baseline Simulator' for the Rocket system being modeled in gXR5. The floating points units utilise DSP slices, hence the latency of these functional units depend on synthesis optimisations performed by the Vivado HLS tool. The following section validates the functional units and the branch predictor models in gXR5.

4.3.1. MinorCPU:Functional Units Latency

The SPFMA, DPFMA and FP Division units DSP48E1 slices as depicted in Figure 4.11 (the full functionality DSP48E1 slice has been depicted in F.1).Since the synthesize-able

verilog does not have macros specifying the synthesis rules to be used, Vivado's default synthesis strategy deploys 'Multiply-Add' and 'Multiply-Accumulate'. With these macros, Vivado maximises the performance by utilizing as many DSP slices as available on the board until there is no improvement in performance. The DSP48E1 slice has a single cycle (shif-add) 25 by 18 bit (operand) multiplier. The output of the multiplier can support a 17-bit shifted output in-order to support larger multiplications. This is followed by a small ALU unit that can be used to carry out the addition operation in a single clock cycle. The registers are absorbed inside the ports of the DSP slices, thereby enabling scheduling of addition and multiply operations depending upon the resource and operand availability as well as the critical path. In case the critical path passes through the DSP slice, the Vivado HLS would break it by pipelining the input ports.

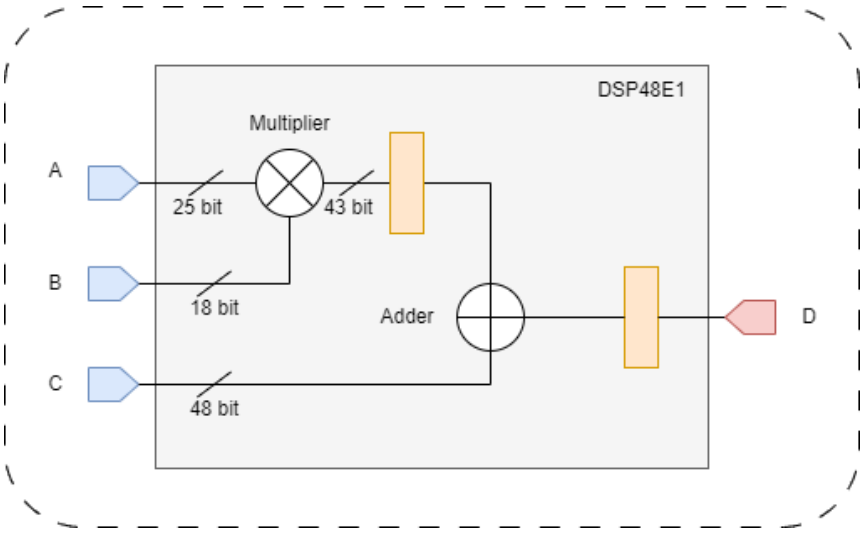


Figure 4.11: DSP48E1 slice functionality

The four ports A, B, C and D can hold the operands for zero to two clock cycle. The figure depicts the ports A and B that can hold the operands in the registers and schedule the operation accordingly. The add operation has to follow one clock cycle after the multiplication as the output of the multiplier is stored in a register and can not be delayed by more than one clock cycle. Similar functionality exists for port C and D.

The Integer Linear Programming Model can be used to find out the minimum possible latency of the floating point units synthesised on the FPGA. This is used as the starting point of the design space exploration done using the CPU class stress-ng benchmarks. The ILP modeling is performed for all the three floating point units. The binary variable X_{ij} represent i^{th} operation scheduled at j^th time step/frame/clock cycle.The ILP modeling constraint equations can be broadly categorised into three [69]:

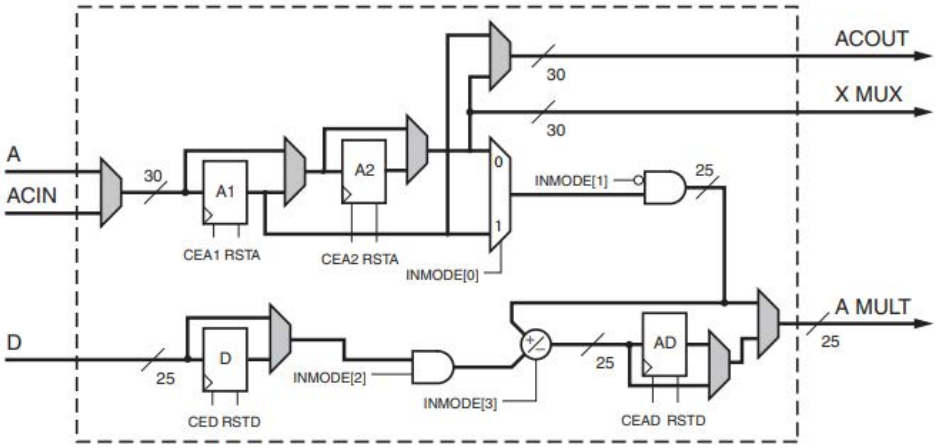


Figure 4.12: 3-stage pipeline ports A and B of DPS48E1 slice [68]

1. Each operation is unique i.e ,

$$\sum_{i=0}^{n} X_{ij} = 1 \tag{4.1}$$

2. The operation dependent on the previous operation can be sequenced only after completion of the previous operation

$$\sum_{l=0}^{\lambda+1} l * X_{il} \geq \sum_{l=0}^{\lambda+1} l * X_{jl} + d_j \tag{4.2}$$

3. The number of operations of a kind scheduled in a particular time frame can be at most equal to the number of available resources of that type.

$$\sum_{m=l-d_i+1}^{l} X_{im} \leq a_k \tag{4.3}$$

The use of ILP modeling is essential to find out possible time-multiplexing of the DSP slices in case of resources of each (or some) type being less than the number of operations of that type. The DSP48E1 slice supports time multiplexing. Moreover, the upper bound on Latency (λ) has been kept arbitrary large.

Estimating Latency of SPFMA unit

The SPFMA uses 2 DSP slices on VC707 FPGA. The MAC (multiply and accumulate) operation performed inside the DSP slice is :

$$Result = A * B + C \tag{4.4}$$

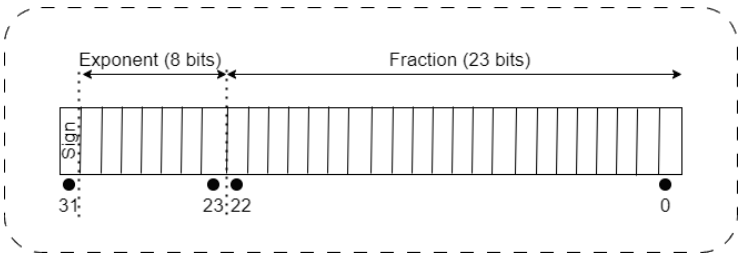


Figure 4.13: Single Precision Floating Point Format IEEE- 745

The operation is commonly known as SAXPY. The operations performed on single precision floating point numbers (depicted in Figure 4.13) can be summarised as below:

- Addition of exponents of A and B (OP ID =1)
- Re-conversion of fraction to number by padding one before the decimal.
- A[23:0] * B[17:0] (multiplication) followed by shift (OP ID =2)
- A[23:0] * B[23:18] (multiplication) followed by shift (OP ID =3)
- addition of the previous two partial multiplication results (OP ID =4)
- addition of significant of C[24:0] post exponent alignment to the result of A * B. (OP ID =5)
- restoring back to single precision FP format

These operations and their dependency has been captured in the Table 4.5. The shifts post multiplications take place in the same clock cycle and so, the ‘multiply’ operation in the Table 4.5 represent multiply and shift operations. Each of the operation has been assigned an Operation (OP) ID along with the OP ID of the source operand on which the Operation depends. The NOP or No-Operation being the ‘source Operation ID’ represent absence of data (or source) dependency. The corresponding constraint equations have been generated using C++ code (refer Appendix G.1) and categorised into the three categories (according to equations 4.1, 4.2 and 4.3. The equations are solved using open source ILP solver [70]. The resulting sequencing graph has been depicted in Figure 4.14. The SPFMA takes a minimum latency of 3 clock cycles to compute the MAC operation.

Estimating Latency of DPFMA unit

The DPFMA utilizes 4 DSP slices on the VC707 FPGA. The DAXPY operation is performed by the DSP slices on double precision number depicted in Figure 4.15.

The list of operations performed on double precision floating point numbers 4.15) can be summarised as below:

- Addition of exponents of A and B (OP ID =1)

Operation ID	Operation Type	Source Operation ID
1	add	NOP
2	multiply	NOP
3	multiply	NOP
4	add	2,3
5	add	4
6	NOP	5

Table 4.5: Single Precision Fused Multiply and Accumulate (SPFMA) operations' dependency

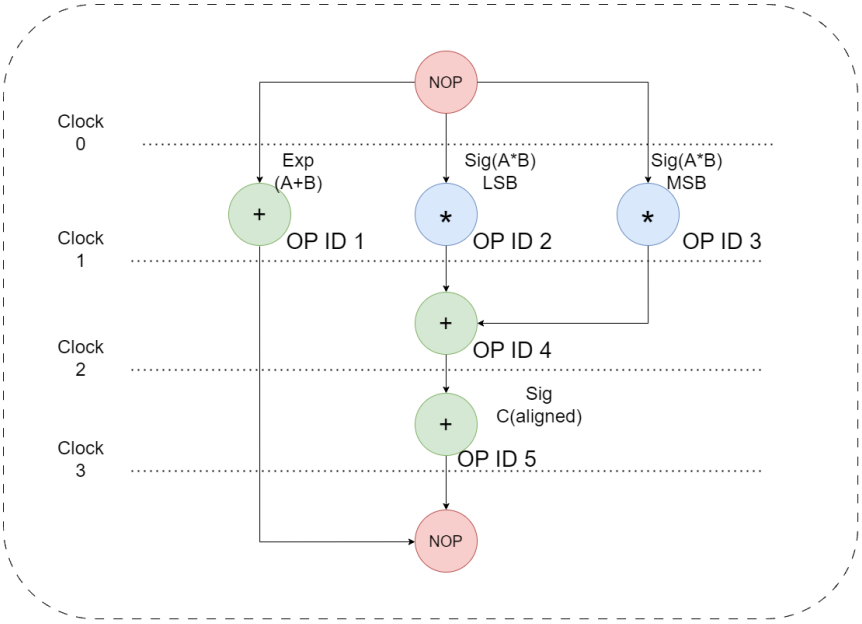


Figure 4.14: Sequencing graph for DPFMA Unit

- Re-conversion of fraction to number by padding one before the decimal.
- A[24:0] * B[17:0] (multiplication) followed by shift (OP ID =2)
- A[24:0] * B[35:18] (multiplication) followed by shift (OP ID =3)
- A[24:0] * B[52:36] (multiplication) (OP ID =4)
- addition of above partial results (OP ID =11,14)
- A[51:25] * B[17:0] (multiplication) followed by shift (OP ID =5)
- A[51:25] * B[35:18] (multiplication) followed by shift (OP ID =6)
- A[51:25] * B[52:36] (multiplication) (OP ID =7)
- addition of the previous partial multiplication (OP ID =12, 15)
- A[52:51] * B[17:0] (multiplication) followed by shift (OP ID =8)
- A[52:51] * B[35:18] (multiplication) followed by shift (OP ID =9)

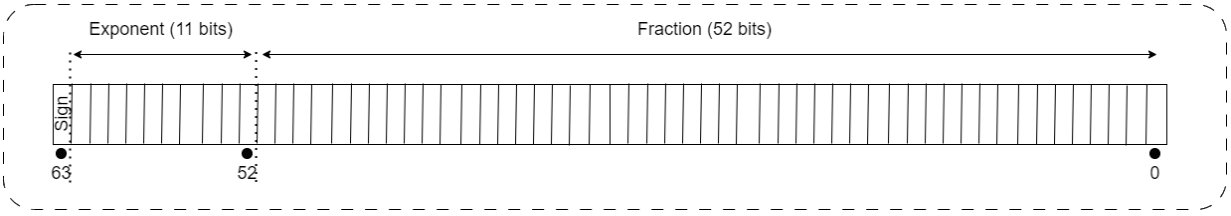


Figure 4.15: Double Precision Floating Point Format IEEE- 745

- A[52:51] * B[52:36] (multiplication) (OP ID =10)
- addition of the previous partial multiplication results (OP ID =13, 16)
- Addition of partial additions (OP ID = 17, 18)
- addition of significant of C[53:0] post exponent alignment to the result of A * B.
- rounding off and restoring back to single precision FP format

The corresponding sequence of operations and their (inter) dependency is summarised in the Table 4.6. The set of operations when put in equation 4.1, 4.2 and 4.3 give the constraint equations (G.2). Solving these set of equations gives the time-frame when each operation is scheduled. The Figure 4.16 depicts the unique scheduling solutions solving these equations.

Estimating Latency of FP Division unit

The FP Division takes place by means of multiplication (division by convergence). The standard algorithm for division by convergence can be summarised by the following equation:

$$Quotient(q) = \frac{Dividend}{Divisor} = \frac{z}{d} = \frac{zx^{(0)}x^{(1)}...x^{(m-1)}}{dx^{(0)}x^{(1)}...x^{(m-1)}} \tag{4.5}$$

As d tends to 1, z tends to q. The multiplier factor x is set to a value of (2-d) for the quadratic convergence of d to 1. For a k bit operand, 2m-1 multiplications and m 2's complement are required where $m = \lceil \log_2(2 \cdot K) \rceil$. For K=53, the number of DSP slices required are 12. However, this is an assumption which needs to be vindicated. The number of DSP slices used by the division unit are in actual 4.

The operations in equation 4.5 and their dependency has been captured in Table 4.7. The corresponding ILP equations are summarised in Appendix G.3.

The solution to the ILP equations lead to the sequencing graph given in Figure 4.17. The minimum possible latency of FP Division unit is 10 clock cycles. The estimated latency of functional units are used as a starting point for the design space exploration. The Table **??** summarises the final latency of various functional units.

Operation ID	Operation Type	Source Operation ID
1	add	NOP
2	multiply	NOP
3	multiply	NOP
4	multiply	NOP
5	multiply	NOP
6	multiply	NOP
7	multiply	NOP
8	multiply	NOP
9	multiply	NOP
10	multiply	NOP
11	add	2,3
12	add	5,6
13	add	8,9
14	add	4,11
15	add	7, 12
16	add	10, 13
17	add	14, 15
18	add	16 17
19	NOP	18

Table 4.6: Double Precision Fused Multiply and Accumulate (DPFMA) operations' dependency

Operation ID	Operation Type	Source Operation ID
1	add	NOP
2	multiply	1
3	multiply	1
4	add	3
7	multiply	2,4
8	multiply	5,6
9	add	7
10	multiply	9
11	multiply	8,9
12	add	10
13	multiply	11,12
14	add	13
15	NOP	14

Table 4.7: Division by Convergence operations and the operation dependency

4.3.2. MinorCPU: Branch Predictor

The existing branch predictor models in gem5 (as described in sections 3.2.3) do not have any branch predictor model that can give similar predictions as that of the Gshare branch predictor in rocket core. Hence, a Gshare branch predictor model has been implemented in gXR5. The branch predictor model is compatible with all CPU models in gem5. The Gshare branch predictor model's attributes (such as 'number of counter bits', 'size of global history register' etc.) can be parameterized via the python configuration scripts. The corresponding changes have been made in scons scripts that sources the new branch predictor model in gXR5. A high level view of the implemented branch predictor is given in Figure 4.18. Compared to the actual branch predictor (refer Section 4.3.2), the modeled branch predictor uses XORs as hash to index into 2-bit counters used for way prediction. The variable gloabHistoryIdx is the index calculated based on hash of branchAddr(the PC) and globalHostoryReg (the global history). The 'tid' represents the thread ID as the CPU models are mutli-threaded. The branch is predicted (represented by Boolean final_prediction) to be taken if the counter value is greater than the localThreshold ('weakly not taken' in case of a 2-bit counter). The following code describes the core functionality of the Gshare branch predictor modeled in gXR5. The complete branch predictor model code is given in Appendix C.2 including the updating the BTB, global history table and squashing incase of mis-prediction.

Listing 4.1: Gshare Branch Predictor Model

```
bool
GshareBP::lookup(ThreadID tid, Addr branchAddr, void * &bpHistory)
{
    unsigned globalHistoryIdx = (((branchAddr >> instShiftAmt)
                                  ^ globalHistoryReg[tid])
                                  & globalHistoryMask);

    assert(globalHistoryIdx < globalPredictorSize);
    bool final_prediction = globalCtrs[globalHistoryIdx]> localThreshold;
    BPHistory *history = new BPHistory;
    history->globalHistoryReg = globalHistoryReg[tid];
    history->finalPred = final_prediction;
    bpHistory = static_cast<void*>(history);
    updateGlobalHistReg(tid, final_prediction);

    return final_prediction;
}
```

The mis-prediction rate for the Gshare and Tournament branch predictor models is depicted in Figure 5.8. The mis-prediction rates are considerably higher than the baseline tournament branch predictor. However, **the MAPE in IPC for the stress-ng stressors reduced from 30% to 18.9% by implementing the new Gshare branch predictor model.**

The MinorCPU model was calibrated using stress-ng benchmarks. The floating point functional units synthesised on the FPGA were fine-tuned to reduce IPC error in simulator. Yet again, implementing the new branch predictor model reduces the error in simulator by more than 10 % for stress-ng benchmarks.

4.4. Concluding remarks

The chapter provided a detailed description of using stress-ng CPU class stressors to calibrate the in-order MinorCPU model against the two target hardwares, the Sifive High-five Freedom Unleashed and the Rocket system emulated on VC707 FPGA. The lack of micro-architectural details about the former target contribute to ‘specification errors’ in the simulator. These were overcome largely by applying the proposed methodology of “component level calibration”. In case of rocket core, the synthesis and implementation optimizations done by the HLS tool contribute to the ‘specification errors’. The ILP

modeling was was used to reduce the design space while calibrating the floating point functional unit latency as these were affected by the HLS tool's synthesis optimisations. The proposed methodology once again proves its soundness by reducing the performance disparity between simulator and hardware. Apart from these, the lack of Gshare branch predictor contributed to 'modeling errors' which were overcome by making the corresponding branch predictor model in gXR5.

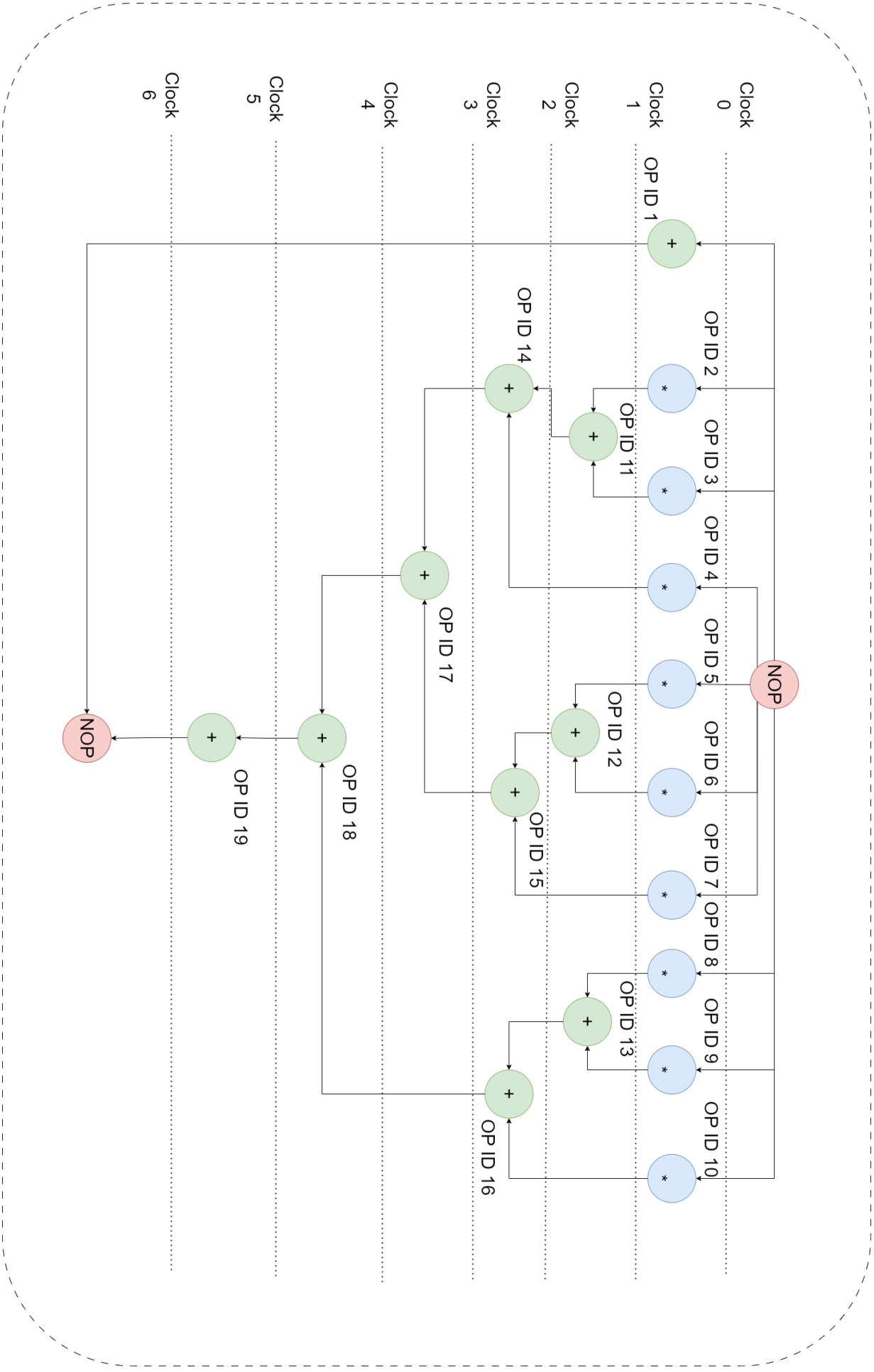


Figure 4.16: Sequencing graph for DPFMA Unit

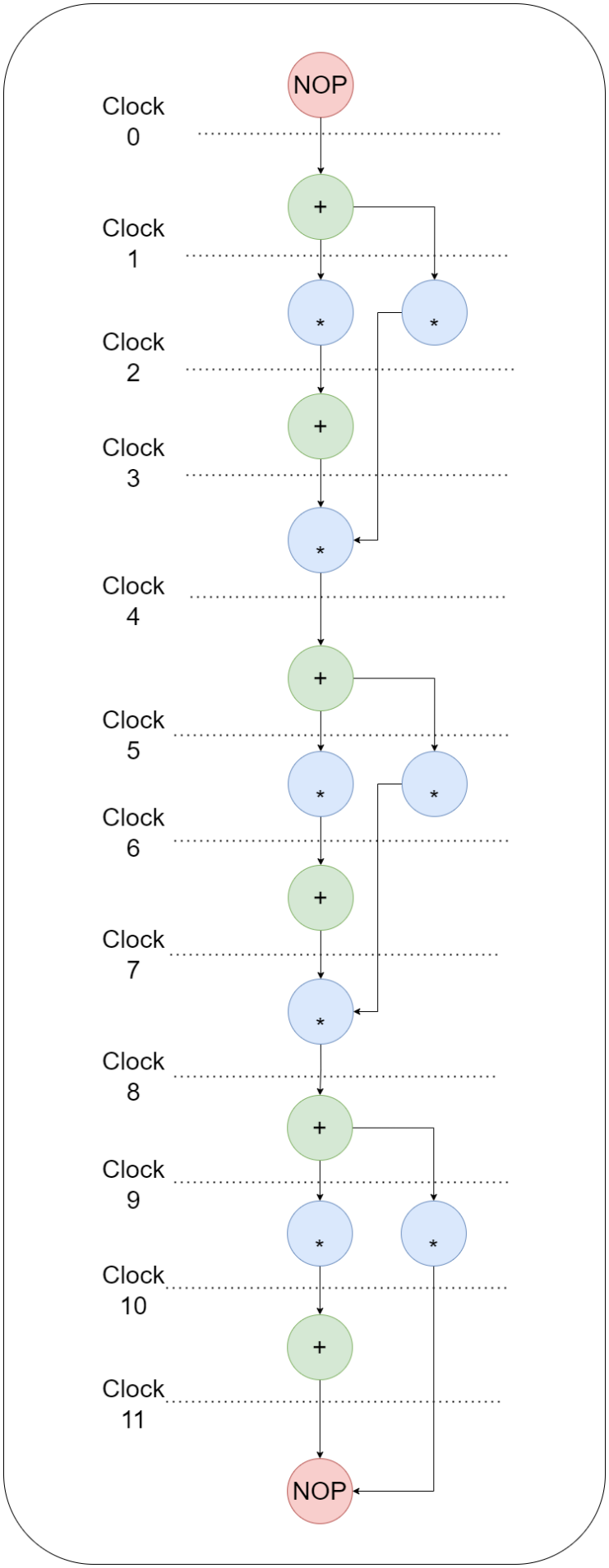


Figure 4.17: Sequencing graph for FP Division Unit

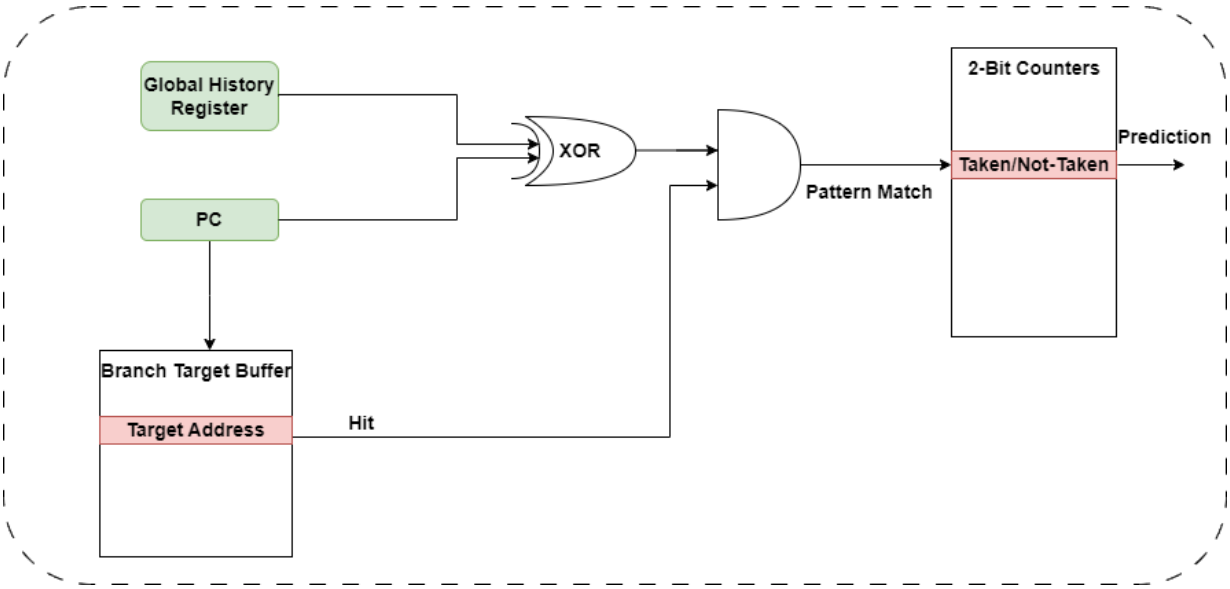


Figure 4.18: Gshare Branch Predictor model high level view

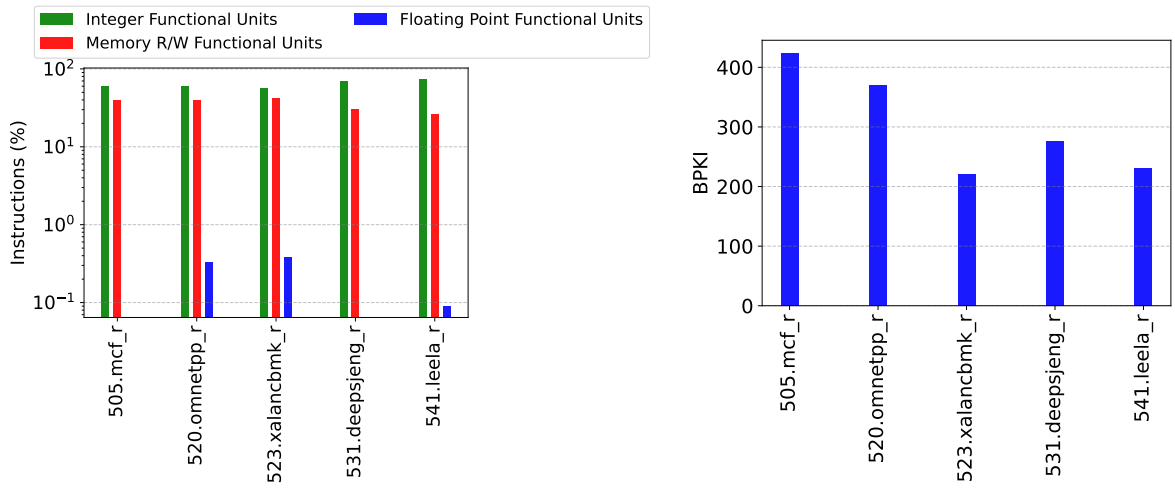
5

Results and Discussion

The intended objective of a full-system simulator is to provide a platform for architects to run real user-space applications that helps quantify the performance gain post architectural improvements. Hence, it is necessary that the performance of the simulator is gauged by running SPEC benchmark suite. SPEC2017 [71] int rate applications are run on the simulator and the target hardware. The chapter begin with characterisation of SPEC2017 applications using gXR5 and subsequently analyses the performance validation results.

The execution time of the application gives an estimate of performance of the simulated system. Hence, the execution time is chosen as the Figure of merit to test the validated simulator while executing SPEC suite applications. However, this is possible only if the simulator and hardware operate at the same frequency. Otherwise, clock cycles elapsed while executing applications is used as an alternative figure of merit.

The software stack of gXR5 full-system simulator includes OpenSBI bootloader, Linux kernel v5.8 and buildroot file system. The applications to be run on gXR5 are cross-compiled (x86 to RISC-V ISA) and put on the disk image used inside gXR5. Once the init process starts, the contents of the python/bash scripts (guest scripts) that launch the application execution inside gXR5 are read and placed in a temporary file. The commands in this temporary file are then given to command terminal inside gXR5. There are python scripts (host) that launch the gXR5 simulator (in full-system mode).Since gXR5 does not support openMP (i.e, one instance of the simulator runs on a single core), multiple instances of the simulator are launched on each host core (manually, via multiple python scripts) to execute multiple applications simultaneously.



(a) Functional Unit Utilisation

(b) Conditional Branch Instructions

Figure 5.1: Profiling SPEC2017 benchmarks for (a) functional unit utilisation and (b) branch incidence.

5.1. Profiling SPEC2017 benchmarks

The SPEC2017 suite is divided into two categories, speed and rate. The SPECspeed suite always run one copy of each benchmark contrary to the SPECrate suite that runs multiple concurrent copies of the benchmark, measuring throughput with OpenMP (pragmas) disabled. For practical reasons, SPECrate suite applications (refer Appendix E) with test input were executed to measure the run/execution time using linux *time* command. The simulation time can be as large as one week for applications such as '531.deepsjeng_r' and less than 3 hours for '523.xalancbmk_r' with the test inputs.

The selected SPEC benchmark applications were profiled using gXR5. Figure 5.1(a) depicts the percentage instructions using integer, floating point and memory read/write functional units. All the applications have negligible floating point operations. However, the load/store instructions are considerably higher (30 to 40%) compared to the selected stress-ng CPU class stressors which had at most 30% load/store instructions (Figure 4.2). The memory models will be the biggest source of specification errors.

Furthermore, the baseline gXR5 is used to profile the applications for intensity of branch incidence. Figure 5.1 depicts the the branch intensity. All of the SPEC suite applications have significant conditional branch instructions with '505.mcf_r' having the most (more than 400 Branches per Kilo Instructions (BPKI)). Hence, it was fair to expect that fine-tuning branch predictor would significantly reduce the IPC/run time errrors in simulator.

5.2. Validated Simulator

The simulator with MinorCPU and L1 level caches calibrated for the target hardware using stress-ng benchmarks is now called the validated simulator. The stress-ng benchmark suites can be considered as the ‘Training’ suite and SPEC applications as the ‘Test’ suite. The SPEC suite is executed in gXR5 only once, thereby saving significant simulation time. Following sections discuss the performance validation results for the two simulator configurations.

5.2.1. Target: Sifive Unleashed

The gXR5 was configured to match the (restricted) available hardware specifications. The resulting configuration is called the ‘baseline’ simulator. On the other hand, post running stress-ng benchmarks to fine tune the MinorCPU model, the simulator is called the ‘validated simulator’.

Stress-ng

The Figure 5.4 depicts the IPC for baseline, calibrated and the hardware running the CPU class stressors. The mean error for stress-ng benchmarks is 14.8%, with 8 out of 10 stressors having less than 10% mean error. The stress-ng benchmarks were used as a test set to calibrate the in-order MinorCPU model. The calibration of Functional Units reduces the error by more than 5%. Subsequently, the design space exploration for selecting the Branch Predictor leads to reduction of in simulator error. The attributes that affect the front end of the pipeline (and control hazards) such as Fetch2 to Fetch 1 delay were also fine-tuned. The Figure 5.2 captures the reduction in MAPE in IPC at different stages of calibration. The ‘post-BP calibration’ includes both the change of branch predictor model from Tournament to Multiperspective Perceptron and the above mentioned attribute. The final stages include calibrating L1 caches and widening the input buffers of the execute stage to reduce performance disparity between hardware and the simulator.

To confirm that the memory indeed is the biggest source of errors, memory class stressors were executed. The functionality of these stressors can be found in Appendix D. While most of the memory class stressors are compatible with x86 systems, membarrier, memcpy, memfd, mlock and mmap are the only stressors compatible with RISC-V systems. The Figure 5.3 shows MAPE in IPC of 75% for the memory stressors. The error is as large as 245 % for memcpy stressor. Moreover, the memory stressors are

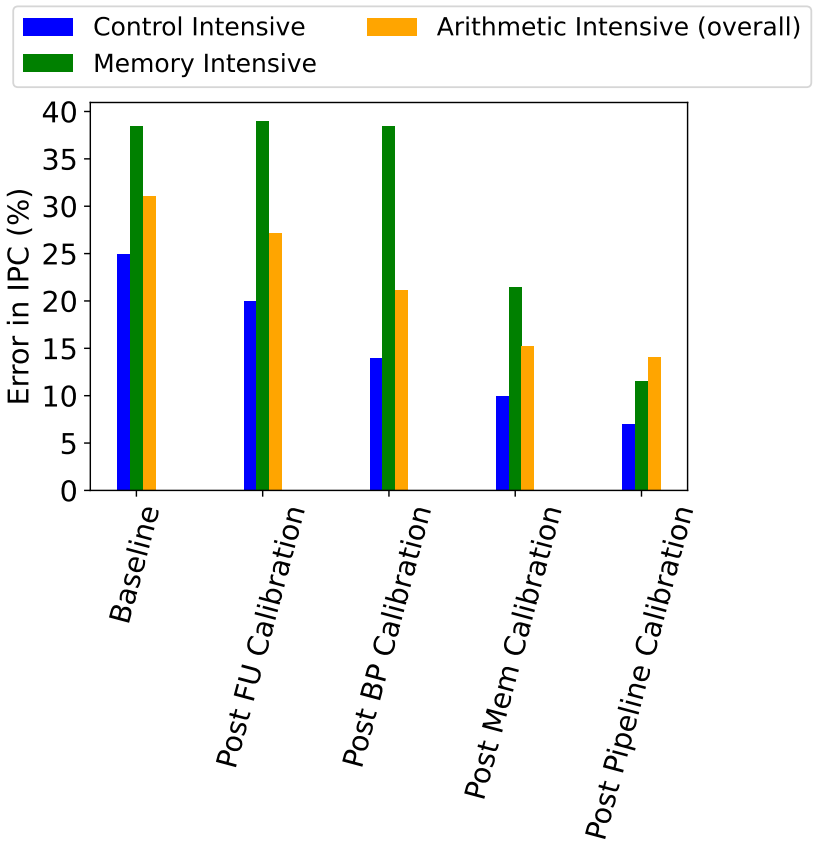


Figure 5.2: Error in IPC at different stages of validation against Sifive Unleashed

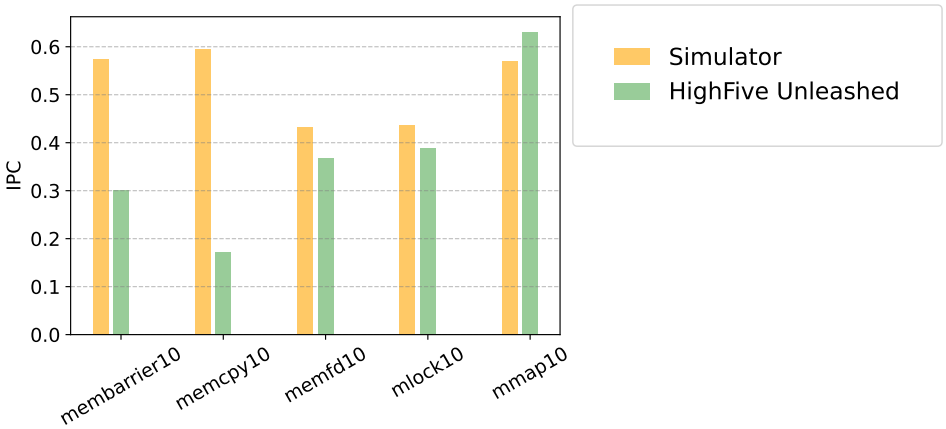


Figure 5.3: Sifive Unleashed vs. Simulator IPC for selected memory stressors

overestimated by the simulator, thereby confirming that overestimation in case of SPEC suite is also due to the memory hierarchy not being fine-tuned. A holistic approach needs to be taken to calibrate the memory (including PTWs, TLBs, Interconnects, etc.). This has been left as future work.

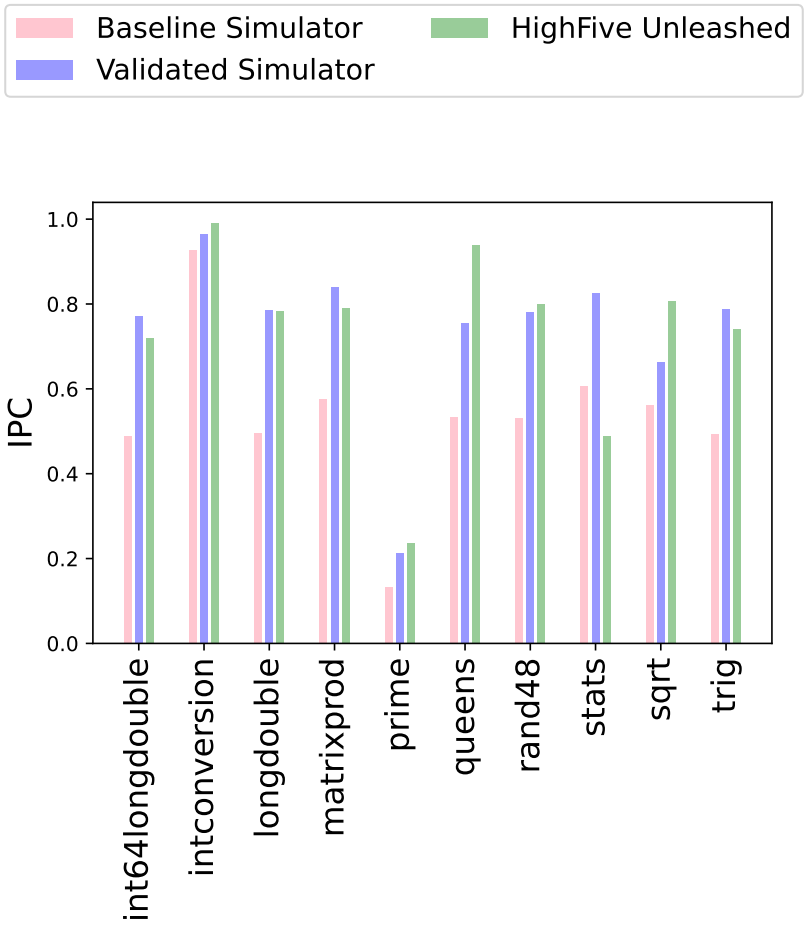


Figure 5.4: Sifive Unleashed vs. Simulator IPC for stress-ng benchmarks

SPEC suite

The run time of the SPEC applications are compared for the baseline, validated simulator and the hardware. Figure 5.5 depicts baseline error of 44% and the final (validated simulator) error of 23.9%. The specification error is brought down by more than 20% just by calibrating the Minor CPU model in gXR5.

As expected, the three application having higher load/store instructions (mcf, omnetpp and xalancbmk) have higher run time error. For the other two applications, the run time error is mere 3.4% and 1.7%, highlighting superior results of the proposed methodology.

The methodology leads to significant decrease in the specification errors of the simulator, without trying to mimic the micro-architectural events (say cache misses, mis-predictions, etc) of the actual hardware. The following section discusses the results while extending the methodology for Rocket system emulated on VC707 FPGA.

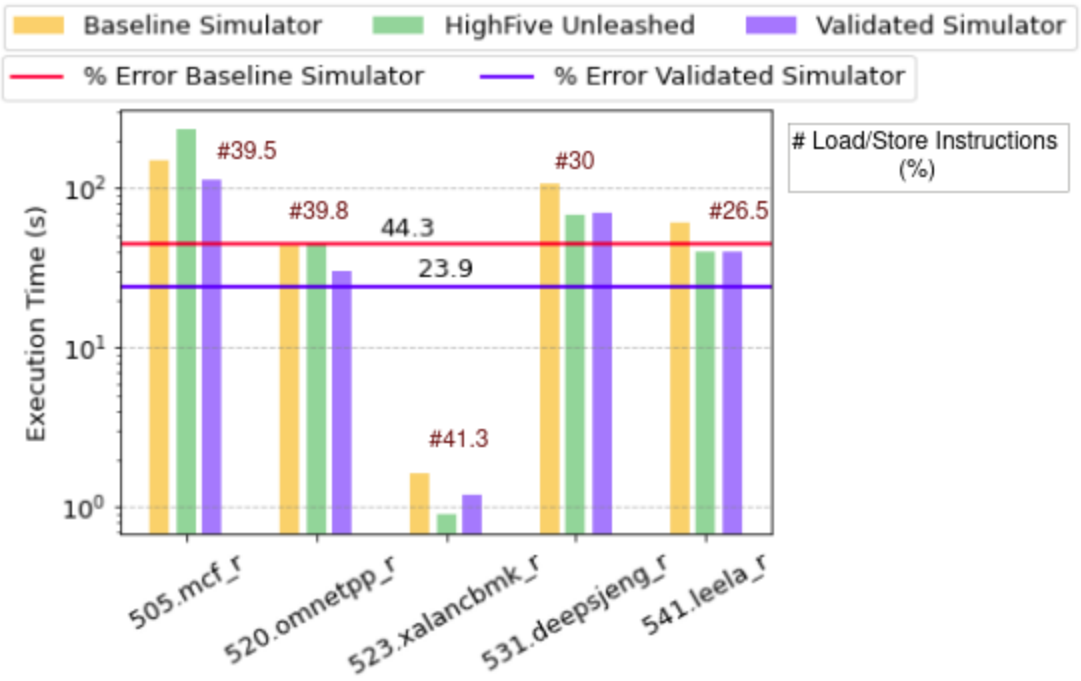


Figure 5.5: Sifive Unleashed vs. Simulator IPC for selected SPEC2017 benchmarks

5.2.2. Target: Rocket System

The Rocket system emulated on VC707 FPGA runs at a frequency different than the system modeled in gXR5. Table 3.1 summarises the technical specifications. The number of clock cycles elapsed while executing SPEC suite applications is used to compare the performance statistics of the hardware and gXR5. The run time of the SPEC applications is extracted using *time* command as earlier and divided by the CPU frequency to get the clock cycles elapsed. Moreover, the ratio of DRAM Controller frequency and CPU operating frequency is kept at 10.665:1 for both Hardware and gXR5. The CPU frequency of the emulated Rocket core can not be further increased (because of negligible positive slack). The DRAM DDR3_8x8 model in gXR5 operates at a fixed frequency of 1.0665 GHz.

Stress-ng

The baseline simulator has the default tournament branch predictor with matching BTB and BHT sizes. Moreover, the floating point functional units have not been validated for the baseline simulator as their synthesis on the FPGA is not known. The IPC of the stressors run on Rocket system emulated on FPGA and gXR5 is depicted in Figure 5.6. The baseline simulator overestimates the IPC for control intensive benchmarks. The MAPE in IPC post floating point functional unit calibration and implementation of Gshare

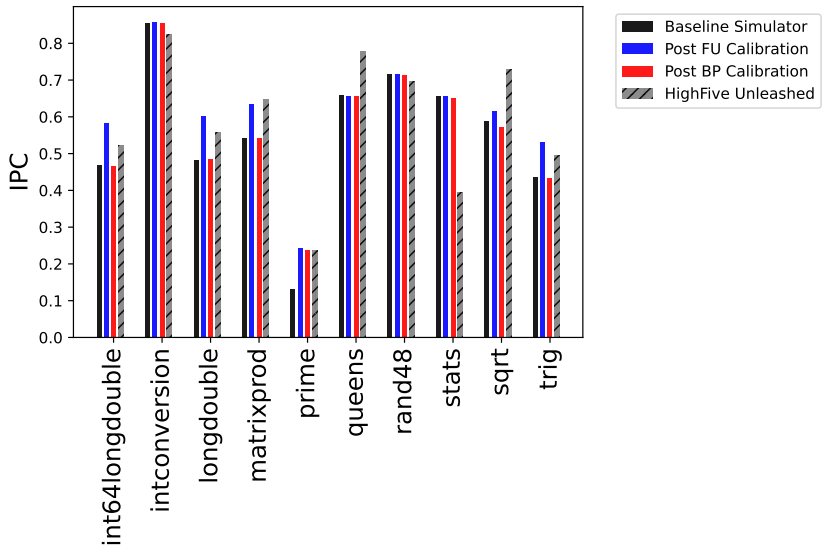


Figure 5.6: Rocket System vs. Simulator IPC for stress-ng benchmarks

branch predictor model is given in Figure 5.7. Implementing Gshare branch predictor model leads to reduction of more than 5 % error in IPC for stress-ng benchmarks. The overestimation of IPC can be attributed to aggressive branch prediction rates of the tournament branch predictor compared to the Gshare branch predictor (Figure 5.8).

SPEC suite

The Figure 5.10 depicts the mis-predcition rates of Tournament and Gshare branch predictor models in gXR5 for SPEC benchmark applications. The Gshare branch predictor leads to more than 10% reduction in error for both stress-ng and SPEC suite. The calibrated simulator has a MAPE in IPC of 16.2% in stress-ng benchmarks. The mean absolute error in clock cycles simulated and executed on hardware is 18.9%.

A single run of simulation for stress-ng benchmarks takes less than 180 minutes to execute stressors for 5 seconds. No more than 15 simulation runs were required to calibrate the Minor CPU model. Many of these simulations were run in parallel, thereby reducing the calibration effort time significantly. The SPEC suite was run only once (after the calibration was done). Moreover, the only HPCs used were instructions and clock cycles (for calculating IPC), thereby significantly reducing the effort spent in characterizing the hardware. Overall, the simulation time required from training phase till test phase of the simulator is considerably less than the existing methodologies.

The MAPE in IPC for the test set is close to the error in execution time of training set for both the target hardware. This is unlike the existing methodology wherein micro-architectural calibration leads to considerably low errors in IPC for synthetic micro-

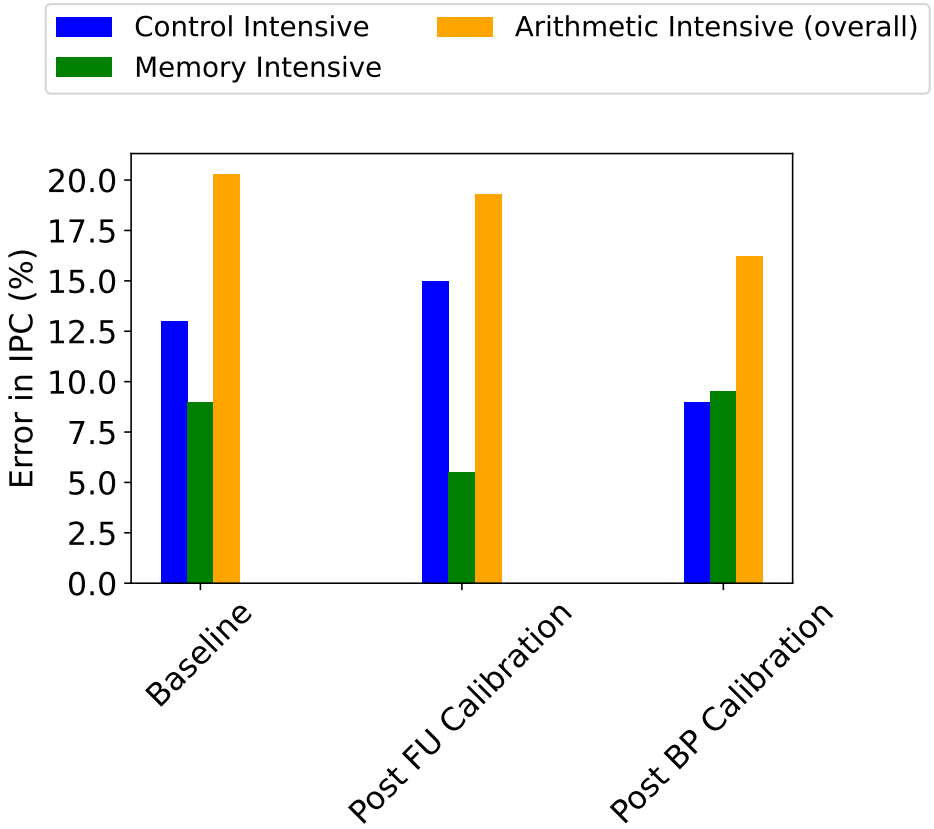


Figure 5.7: Error in IPC at different stages of validation against Rocket system

benchmarks (6%) but relatively high error for SPEC/PARSEC benchmark applications (20 %) [48]. The superiority of the the component level calibration methodology owe to the higher order time complexity of the training set compared to the synthetic micro-benchmarks.

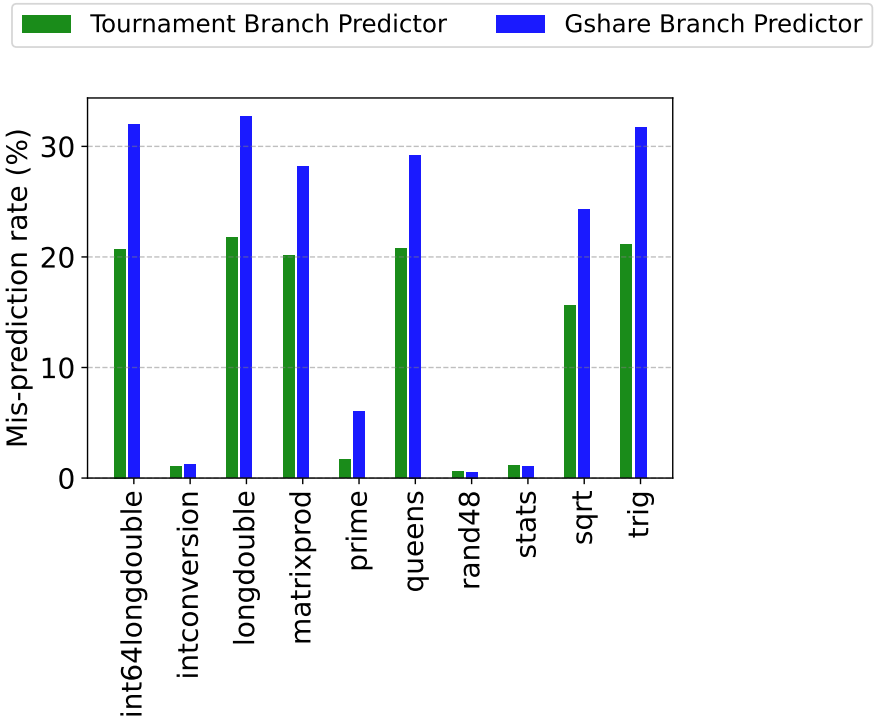


Figure 5.8: Branch mis-prediction rates for Tournament and Gshare branch predictor for stress-ng benchmarks

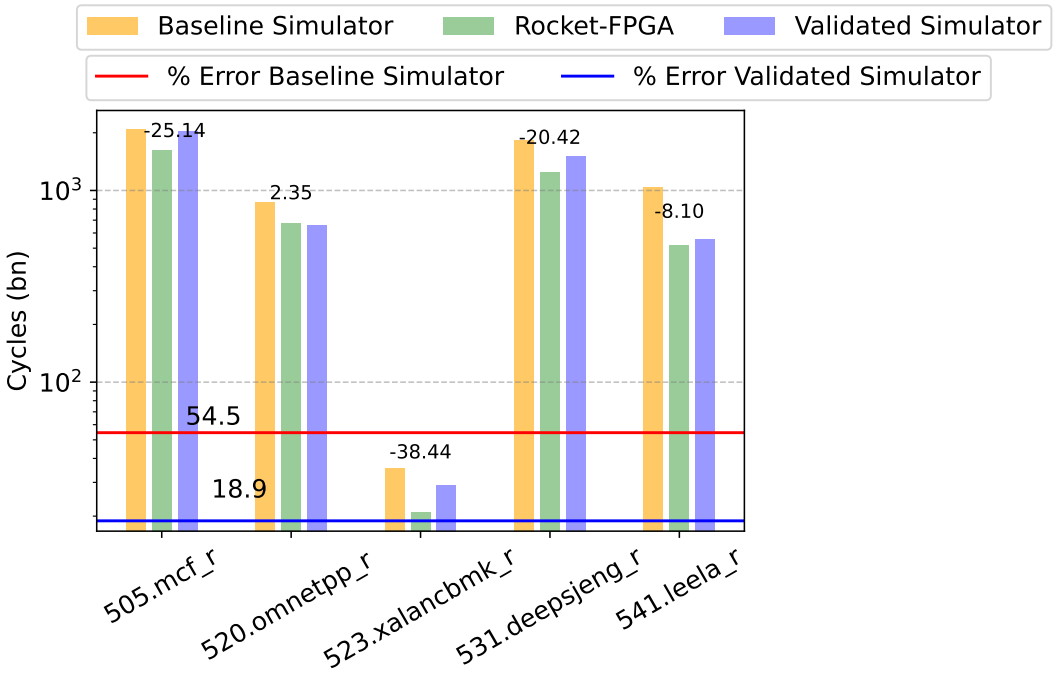


Figure 5.9: Rocket System vs. Simulator IPC for selected SPEC2017 benchmarks

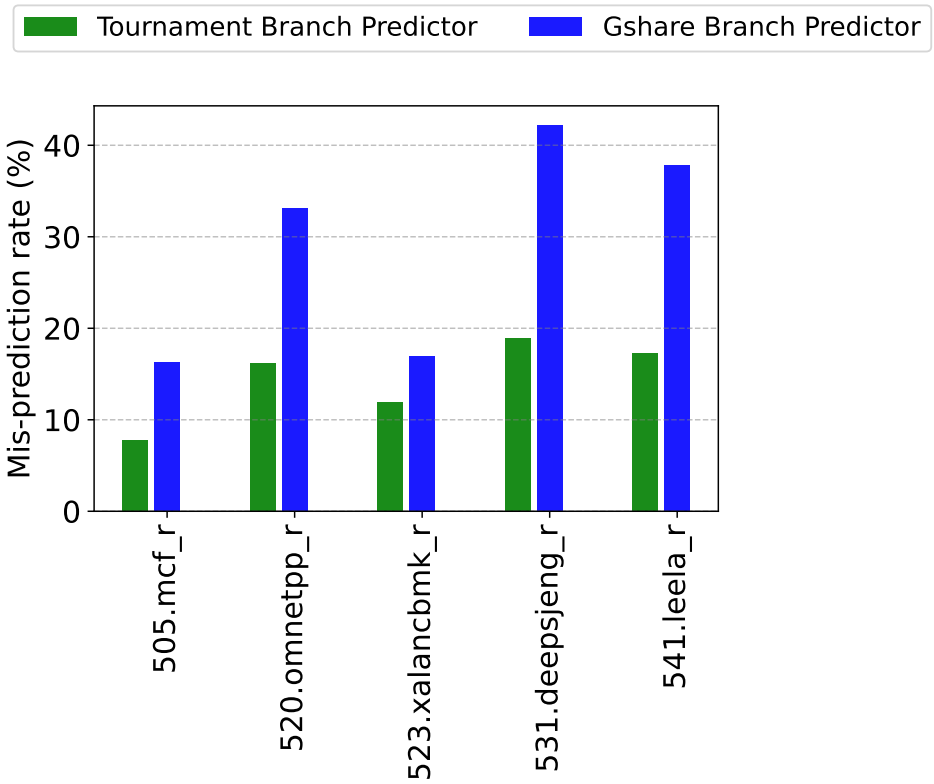


Figure 5.10: Branch mis-prediction rates for Tournament and Gshare branch predictor models for SPEC2017 applications

6

Conclusion

"The only way to do great work is to love what you do"
-Steve Jobs

6.1. Refining the proposed methodology

The proposed methodology of 'component level calibration' with stress-ng as the training set achieves the following objectives:

- It streamlines the design of micro-benchmarks for calibration of simulator models using Hardware Performance Counters (HPCs). The only other attempt to streamline the design of micro-benchmarks has been for memory calibration [53].
- It provides a generic methodology that can be extended for Out-of-Order (OoO) core models in gem5 or other architectural simulators. The methodology is not specific to RISC-V ISA and would be equally effective for other ISAs.
- The simulation time spent on validation of architectural simulators can be reduced significantly. This is in contrast to direct 'validation by inspection' methodology wherein the macro benchmarks are run both as training and/or test sets.
- The need to characterize hardware is significantly reduced, thereby providing a perfect methodology for system-level engineers/simulator designers to validate simulators rather quickly.

However, the methodology needs to be extended for other component calibration. A holistic approach needs to be taken for modeling and fine-tuning I/O, Memory and

Interconnects. A good ground to test the methodology would be to simulate multi-core systems with shared memory. Ruby caches in gem5 provide a much detailed micro-architecture including various cache coherency fabrics. The stress-ng stressors can be run on multiple cores to tune the modeled memory hierarchy.

A notable drawback of the methodology can be over-fitting for stress-ng benchmarks. Errors in simulator for stress-ng benchmarks can be reduced by fine-tuning various (permutations of) attributes. For example, instead of fine-tuning branch predictor model, the designer can fine-tune the input buffer size of the execute unit and allow execution across multiple clock cycles, thereby achieving similar simulator performance. This has the potential to reduce errors for control intensive micro-benchmarks as the control hazards have been mitigated by achieving a high functional unit utilisation. An ideal approach would be to fine-tune the branch predictor. However, in such a case, the errors in macro-benchmarks are observed to be very high (even though they are <10% for stress-ng benchmarks). Hence, brute-forcing the fine-tuning without understanding (and to an extent reverse-engineering) micro-architecture of the hardware can lead to sub-par performance validation results.

The benefits of the proposed methodology over weigh its drawbacks. Considerable effort and time can be saved in validating simulators.

6.2. Reflecting on the Results

The validation results are based on 5 out 10 benchmark applications of integer rate SPEC2017 suite. Citron [72] has carried out a survey of three high impact architectural conferences (ISCA, HPCA and Micro). Out of 173 published papers, only 23 use the entire SPEC suite (mostly SPEC CPU2000 [49], since the survey was carried out in 2003). In case of architectural simulators, executing the entire suite is not possible for practical reasons. The simulator throughput is often around few hundred of KIPS (with detailed CPU models) or few MIPS (for say, Timing CPU model in gem5).

A common practice is to have ‘reduced input set simulation’ [72] wherein test inputs can be used or new inputs can be designed to have shorter execution times for benchmarks. Other alternative is to have ‘truncated execution simulation’ wherein a (contiguous) part of benchmark application is executed (say first z million instructions). However, this technique has the disadvantage of being under or non-representative of computational complexity of the original benchmark application. Variations of this approach have been proposed to make truncated benchmarks more representative. Another common practice is to sample the benchmark application. These can be ‘representative sampling

simulation', 'periodic or random sampling simulation', etc., depending upon the sampling strategy/interval.

The use of test inputs for SPEC2017 suite goes with the existing practice, especially when validating architectural simulators. Mean Absolute Percentage Error of 22.9 %and 18.3 % in simulator against real hardware is at par with existing performance validation results of full-system simulators.

References

[1] *Spike Simulator*. Accessed on 28.07.2023, [Online]. Available https://wiki.riscv.org/display/HOME/Emulators+and+Simulators#EmulatorsandSimulators-Spike/riscv-isa-sim.

[2] *QEMU*. Accessed on 28.07.2023, [Online]. Available https://www.qemu.org/.

[3] T. Austin et al. “Simplescalar: An Infrastructure for Computer System Modeling”. In: *Computer* 35.2 (2002), pp. 59–67. DOI: 10.1109/2.982917.

[4] Nathan Binkert et al. “The GEM5 simulator”. In: *ACM SIGARCH Computer Architecture News* 39.2 (2011), pp. 1–7. DOI: 10.1145/2024716.2024718.

[5] Milo M. Martin et al. “Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset”. In: *ACM SIGARCH Computer Architecture News* 33.4 (2005), pp. 92–99. DOI: 10.1145/1105734.1105747.

[6] Matt T. Yourst. “PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator”. In: *2007 IEEE International Symposium on Performance Analysis of Systems Software* (2007), pp. 23–34. DOI: 10.1109/ISPASS.2007.363733.

[7] Jason Cong et al. “PARADE: A cycle-accurate full-system simulation Platform for Accelerator-Rich Architectural Design and Exploration”. In: (2015), pp. 380–387. DOI: 10.1109/ICCAD.2015.7372595.

[8] N.L. Binkert et al. “The M5 simulator: Modeling Networked Systems”. In: *IEEE Micro* 26.4 (2006), pp. 52–60. DOI: 10.1109/mm.2006.82.

[9] *Leading Semiconductor Industry Players Join Forces to Accelerate RISC-V*. Accessed on 10.08.2023, [Online].Available https://www.qualcomm.com/news/releases/2023/08/leading-semiconductor-industry-players-join-forces-to-accelerate.

[10] *LStreamline for gem5*. Accessed on 10.08.2023, [Online].Available https://developer.arm.com/tools-and-software/embedded/legacy-tools/ds-5-development-studio/streamline/streamline-for-gem5.

[11] *gXR5: A gem5-based full-system RISC-V simulator (WiPLASH)*. Accessed on 28.01.2023, [Online].Available https://www.wiplash.eu/WiPLASH_D5.1_appendix.pdf.

[12] *HiFive- Freedom Unleashed*. Accessed on 28.01.2023, [Online].Available https://www.sifive.com/boards/hifive-unleashed.

[13] *Rocket Chip*. Accessed on 03.08.2023, [Online].Available https://github.com/chipsalliance/rocket-chip.

[14] *AMD Virtex 7 FPGA VC707 Evaluation Kit*. Accessed on 10.08.2023, [Online].Available https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html.

[15] Alexander Dörflinger et al. “A comparative survey of open-source application-class RISC-V processor implementations”. In: *Proceedings of the 18th ACM International Conference on Computing Frontiers* (May 2021), pp. 12–20. DOI: 10.1145/3457388.3458657.

[16] *1666-2011 - IEEE standard for standard systemc language reference*. Accessed on 28.01.2023,[Online].Available https://ieeexplore.ieee.org/document/6134619.

[17] *Ovpsim Instruction Set Simulator*. Accessed on 28.01.2023, [Online]. Available https://www.ovpworld.org/technology_ovpsim.

[18] *The ant architecture–an architecture for CS1*. Accessed on 10.08.2023, [Online].Available https://dash.harvard.edu/bitstream/handle/1/25620472/tr-13-98.pdf.

[19] *CPU Sim Home Page*. Accessed on 10.08.2023, [Online].Available https://cs.colby.edu/djskrien/CPUSim/.

[20] *CLa Máquina Rudimentaria (MR) es un procesador pedagógico.* Accessed on 10.08.2023, [Online].Available https://docencia.ac.upc.edu/eines/MR/.

[21] Arun Rodrigues et al. “The structural simulation toolkit”. In: *ACM SIGMETRICS Performance Evaluation Review* 38.4 (Mar. 2011), pp. 37–42.

[22] Daniel Sanchez et al. “Zsim:fast and accurate microarchitectural simulation of thousand-core systems”. In: *Proceedings of the 40th Annual International Symposium on Computer Architecture* (June 2013), pp. 475–486. DOI: 10.1145/2485922.2485963.

[23] Trevor E. Carlson et al. “Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation”. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov. 2011), pp. 1–12. DOI: 10.1145/2063384.2063454.

[24] Caroline Collange et al. “Barra: A Parallel Functional Simulator for GPGPU”. In: *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (2010). DOI: 10.1109/mascots.2010.43.

[25] David Wang et al. “DRAMsim: A memory system simulator”. In: *ACM SIGARCH Computer Architecture News* 33.4 (2005), pp. 100–107. DOI: 10.1145/1105734.1105748.

[26] Yoongu Kim et al. “Ramulator: A fast and extensible dram simulator”. In: *IEEE Computer Architecture Letters* 15.1 (2016), pp. 45–49. DOI: 10.1109/lca.2015.2414456.

[27] Marco Antonio Alves et al. “Sinuca: A validated Micro-Architecture Simulator”. In: *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems* (2015). DOI: 10.1109/hpcc-css-icess.2015.166.

[28] Sheng Li et al. “McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures”. In: *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2009), pp. 469–480.

[29] D. Brooks et al. “Wattch: A framework for architectural-level power analysis and Optimizations”. In: *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)* (2000), pp. 83–94. DOI: 10.1109/isca.2000.854380.

[30] Mircea R. Stan et al. “Hotspot: A dynamic compact thermal model at the processor-architecture level”. In: *Microelectronics Journal* 34.12 (2003), pp. 1153–1165. DOI: 10.1016/s0026-2692(03)00206-4.

[31] Mendel Rosenblum et al. “Using the Simos Machine Simulator to study complex computer systems”. In: *ACM Transactions on Modeling and Computer Simulation* 7.1 (1997), pp. 78–103. DOI: 10.1145/244804.244807.

[32] Phillip Stanley-Marbell et al. “Sunflower: Full-system, embedded microarchitecture evaluation”. In: *High Performance Embedded Architectures and Compilers* (2007), pp. 168–182. DOI: 10.1007/978-3-540-69338-3_12.

[33] Avadh Patel et al. “MARSS: A full system simulator for x86 cpus”. In: *Proceedings of the 48th Design Automation Conference* (2011), pp. 29–30. DOI: 10.1145/2024724.2024954.

[34] E. Larson et al. “Mase: A novel infrastructure for detailed microarchitectural modeling”. In: *2001 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS.* (Nov. 2001), pp. 1–9. DOI: 10.1109/ispass.2001.990668.

[35] Jose Renau et al. *SESC simulator*. http://sesc.sourceforge.net. Jan. 2005.

[36] Erez Perelman et al. “Using SimPoint for accurate and efficient simulation”. In: *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* 31.1 (June 2003), pp. 318–319. DOI: 10.1145/781027.781076.

[37] Yanqing Zhang et al. “GRANNITE: Graph Neural Network Inference for Transferable Power Estimation”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)* (2020), pp. 1–6. DOI: 10.1109/DAC18072.2020.9218643.

[38] Guillem López-Paradís et al. “Fast Behavioural RTL Simulation of 10B Transistor SoC Designs with Metro-Mpi”. In: *2023 Design, Automation Test in Europe Conference Exhibition (DATE)* (2023), pp. 1–6. DOI: 10.23919/DATE56975.2023.10137080.

[39] Anthony Gutierrez et al. “Sources of error in full-system simulation”. In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2014), pp. 13–22. DOI: 10.1109/ISPASS.2014.6844457.

[40] *SPEC CPU® 2006*. Accessed on 24.07.2023, [Online].Available https://www.spec.org/cpu2006/.

[41] Yasir Mahmood Qureshi et al. “Gem5-X: A Gem5-Based System Level Simulation Framework to Optimize Many-Core Platforms”. In: *2019 Spring Simulation Conference (SpringSim)* (2019), pp. 1–12. DOI: 10.23919/SpringSim.2019.8732862.

[42] Fernando A. Endo et al. “Micro-architectural simulation of in-order and out-of-order arm microprocessors with GEM5”. In: *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)* (2014). DOI: 10.1109/samos.2014.6893220.

[43] Christian Bienia et al. “The parsec benchmark suite”. In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (Oct. 2008), pp. 72–81. DOI: 10.1145/1454115.1454128.

[44] Anastasiia Butko et al. “Accuracy evaluation of GEM5 simulator system”. In: *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)* (2012). DOI: 10.1109/recosoc.2012.6322869.

[45] S.C. Woo et al. “The SPLASH-2 programs: characterization and methodological considerations”. In: (1995), pp. 24–36. DOI: 10.1109/ISCA.1995.524546.

[46] Man-Lap Li et al. “The ALPBench benchmark suite for complex multimedia applications”. In: (2005), pp. 34–45. DOI: 10.1109/IISWC.2005.1525999.

[47] *STREAM benchmarks*. Accessed on 25.07.2023, [Online].Available https://www.cs.virginia.edu/stream/ref.html.

[48] R. Desikan et al. “Measuring experimental error in microprocessor simulation”. In: *Proceedings 28th Annual International Symposium on Computer Architecture* (2001), pp. 266–277. DOI: 10.1109/ISCA.2001.937455.

[49] *SPEC CPU® 2000*. Accessed on 24.07.2023, [Online].Available https://www.spec.org/cpu2000/.

[50] *SiNUCA benchmarks*. Accessed on 24.07.2023, [Online].Available https://github.com/mazalves/sinuca.

[51] Ayaz Akram et al. “Validation of the GEM5 simulator for x86 architectures”. In: *IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)* (2019). DOI: 10.1109/pmbs49563.2019.00012.

[52] Jacob Benesty et al. “Pearson correlation coefficient”. In: *Noise Reduction in Speech Processing* (2009), pp. 1–4. DOI: 10.1007/978-3-642-00296-0_5.

[53] Quentin Huppert et al. “Memory hierarchy calibration based on real hardware in-order cores for accurate simulation”. In: *2021 Design, Automation and Test in Europe Conference and Exhibition(DATE)* (2021). DOI: 10.23919/date51398.2021.9474108.

[54] Irene Wang et al. “Evaluation of GEM5 for performance modeling of ARM cortex-R based embedded socs”. In: *Microprocessors and Microsystems* 93 (2022), p. 104599. DOI: 10.1016/j.micpro.2022.104599.

[55] André Seznec et al. “A case for (partially) tagged geometric history length branch prediction”. In: *The Journal of Instruction-Level Parallelism* 8 (2006), p. 23.

[56] Daniel A. Jiminez. “Multiperspective Perceptron Predictor”. In: *The Journal of Instruction-Level Parallelism* (2016).

[57] André Seznec. “TAGE-SC-L Branch Predictors Again”. In: *5th JILP Workshop on Computer Architecture Competitions (JWAC-5) : Championship Branch Prediction (CBP-5)* (2016).

[58] Alexander Dörflinger et al. “A comparative survey of open-source application-class RISC-V processor implementations”. In: *Proceedings of the 18th ACM International Conference on Computing Frontiers* (May 2021), pp. 12–20. DOI: 10.1145/3457388.3458657.

[59] *Chisel/FIRRTL Hardware Compiler Framework*. Accessed on 16.08.2023, [Online].Available https://www.chisel-lang.org/.

[60] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[61] *VCS Simulator*. Accessed on 03.08.2023, [Online].Available https://www.synopsys.com/verification/simulation/vcs.html.

[62] *RISC-V GNU Compiler Toolchain*. Accessed on 03.08.2023, [Online].Available https://github.com/riscv-collab/riscv-gnu-toolchain.

[63] *Verilator*. Accessed on 03.08.2023, [Online].Available https://www.veripool.org/verilator/.

[64] *Vivado-RISCV*. Accessed on 03.08.2023, [Online].Available https://github.com/eugene-tarassov/vivado-risc-v.

[65] *AMD High Level Design*. Accessed on 03.08.2023, [Online].Available https://www.xilinx.com/products/design-tools/vivado/high-level-design.html.

[66] *Working with XSDB*. Accessed on 03.08.2023, [Online].Available https://www.xilinx.com/htmldocs/xilinx2019_1/SDK_Doc/SDK_concepts/concept_Xilinxsystemdebugger.html.

[67] *RISC-V, Spike, and the Rocket Core*. Accessed on 06.08.2023,[Online].Available https://inst.eecs.berkeley.edu/~cs250/fa13/handouts/lab2-riscv.pdf.

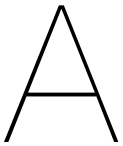
[68] *7 Series DSP48E1 Slice - User Guide*. Accessed on 05.08.2023,[Online].Available https://docs.xilinx.com/v/u/en-US/ug479_7Series_DSP48E1.

[69] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. 1st. McGraw-Hill Higher Education, 1994.

[70] *Introduction to LP solve*. Accessed on 27.07.2023, [Online].Available https://lpsolve.sourceforge.net/5.5/.

[71] *SPEC CPU® 2017*. Accessed on 28.01.2023, [Online].Available https://www.spec.org/cpu2017/.

[72] D. Citron. “MisSPECulation: Partial and misleading use of spec CPU2000 in Computer Architecture Conferences”. In: *ACM SIGARCH Computer Architecure News* 31.2 (May 2003), pp. 52–61. DOI: 10.1109/isca.2003.1206988.



Publications

1. **Karan Pathak**, Joshua Klein, Giovanni Ansaloni, Marina Zapater and David Atienza, ”**Validating Full-System RISC-V Simulator: A Systematic Approach**”, RISC-V Summit Europe, Barcelona, 5-9 June, 2023. (Poster Presentation)
2. **Karan Pathak**, Joshua Klein, Giovanni Ansaloni, Marina Zapater, Georgi Gaydadjiev, David Atienza, ”**A Validated Linux-capable RISC-V Simulator**”, (**Submitted**)

Validating Full-System RISC-V Simulator: A Systematic Approach

Karan Pathak[1,2*], Joshua Klein[1], Giovanni Ansaloni[1], Marina Zapater[1,3] and David Atienza[1]

[1]Embedded Systems Laboratory, École Polytechnique Fédérale de Lausanne
[2]Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology
[3]REDS. School of Engineering and Management Vaud, HES-SO University of Applied Sciences and Arts Western Switzerland

Abstract

RISC-V-based Systems-on-Chip (SoCs) are witnessing a steady rise in adoption in both industry and academia. However, the limited support for Linux-capable Full System-level simulators hampers development of the RISC-V ecosystem. We address this by validating a full system-level simulator, gXR5 (gem5-eXtensions for RISC-V), against the SiFive HiFive Unleashed SoC, to ensure performance statistics are representative of actual hardware. This work also enriches existing methodologies to validate the gXR5 simulator against hardware by proposing a systematic component-level calibration approach. The simulator error for selected SPEC CPU2017 applications reduces from 44% to 24%, just by calibrating the CPU. We show that this systematic component-level calibration approach is accurate, fast (in terms of simulation time), and generic enough to drive future validation efforts.

Introduction

A Linux-capable Full System-level simulator is one that executes benchmarks (e.g. SPEC CPU2017 suite) atop a kernel, system interface, and detailed hardware models built-in software. To date, a full system-level simulator for RISC-V has not been publicly validated against fabricated hardware to target the precise modeling of the execution of benchmark suites. gXR5 is a RISC-V-based Linux-capable, full system-level simulator built into the gem5 architectural simulator that is capable of simulating the run-time of high-level applications [1]. Since it is built on gem5 [2], an open-source, cycle-accurate, and event-driven architecture simulator, it supports simulation of SoCs by providing tunable architectural and micro-architectural models. It can simulate multiple instruction set architectures using ISA-agnostic CPU, bus and memory models.

Although gem5 and gXR5 are easily tunable, validating the simulator for the SPEC CPU2017 benchmark suite is a non-trivial task. The SPEC CPU2017 suite, when run uncalibrated (hereafter 'baseline') on gXR5, has a mean absolute percentage error (simulated vs actual hardware, hereafter, 'error') of 44.3% in execution time. The existing methodologies for validating simulator models for the SPEC suites use synthetic micro-benchmarks [3, 4, 5, 6] to reduce the error (depicted as "micro-architectural-level" calibration in Fig 1). These micro-benchmarks are not representative of time complexity of actual workloads, thereby leading to poor performance accuracy of simulators when real user applications are run. Desikan et al. [3] achieve 18% simulator error in IPC for macro-benchmarks derived from the SPEC CPU2000 benchmark suite. Attempts have been made to use the correlation between micro-architectural events and error in IPC (e.g. Pearson's correlation) [5] and Hardware Performance Counters (HPCs) for validating the simulator. Huppert et al. [4] achieve an error of $20-25\%$ in IPC for the SPEC CPU2017 benchmark suite.

Our proposed methodology introduces component-level calibration of gem5 models targeting the Sifive HighFive Unleashed System-on-Chip (SoC), which is faster (in terms of simulation time), accurate, and extensible to other benchmarks. A CPU model in gXR5 is calibrated using the stress-ng benchmark suite[1], reducing the error in IPC from 36% to 11.8%. Once calibrated, the execution time error reduces from 44% to 24% while running the selected SPEC CPU2017 benchmark suite. We intend to release and open source the validated gXR5 CPU model and supporting materials for adoption by the RISC-V community.

Methodology

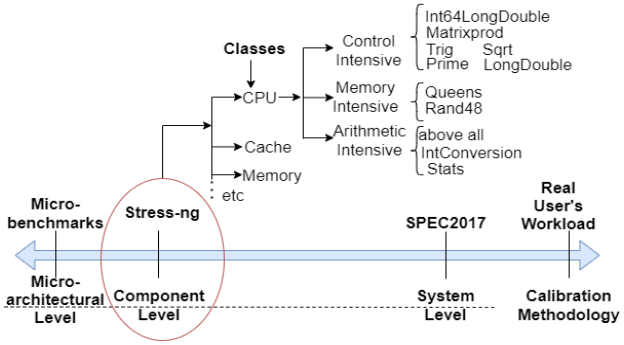


Figure 1: *Component-level calibration using the stress-ng benchmark suite.*

The target of this validation focuses on the CPU model (four-stage MinorCPU pipeline), leaving the tuning of the memory hierarchy for future work. The proposed methodology employs component-level calibration using selected "CPU class" *stressors* (collection of micro-benchmarks) of the stress-ng benchmark suite to target the FU540-C000 in-order CPU design of Linux-capable Sifive HiFive Unleashed SoC. The FU540-C000 (also referred to as U54) is a RV64GC core. The HiFive Unleashed SoC is a quin-core SoC, with one small CPU that supports real-time constraints by hosting RTOS, while the other four U54

*Corresponding Author: karan.pathak@epfl.ch

[1] https://wiki.ubuntu.com/Kernel/Reference/stress-ng

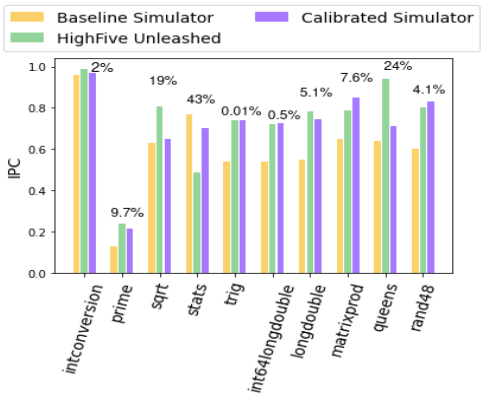


Figure 2: *Performance of Simulator vs HighFive Unleashed SoC for stress-ng benchmarks.*

cores are pipelined in-order processors[2] typically targeting workloads in user space. We base our simulated model and validation effort on single-core core workloads executing on the U54 core.

stress-ng Calibration

The stress-ng suite stresses a particular component or set of components of the system (in this case the CPU) by using *stressors*. The selected CPU class stressors were classified into three categories:

- **Control Intensive:** having complex control structures (eg. nested for-if-else) with at least 100 branches per 1000 instructions.
- **Memory Intensive:** having 20% or higher load/store instructions in the total op-code mix of the program. A maximum 30% instructions are load/store (for "queens" and "rand48").
- **Arithmetic Intensive:** having 20% or more Instructions using Integer/Float functional units.

The most significant attributes of the baseline and calibrated MinorCPU models are summarised in Table 1. Figure 2 depicts the IPC of Baseline, Calibrated simulator and Hardware for stress-ng benchmarks, along with the absolute percentage of IPC error in the Calibrated simulator. The simulator achieves a Mean Absolute Percentage Error in IPC of 11.8%.

Table 1: *Simulator Model- MinorCPU Attributes*

Component	Attribute	Simulated Model	
		Baseline	Calibrated
ReadMemFU	OpLat (cycles)	4	2
IntDivFU	OpLat (cycles)	33	19
Fetch unit	fetch1Tofetch2 BackwardDelay	1 (cycles)	0 (cycles)
Branch-Predictor	Type	Tournament	Multi perspective Perceptron

Experimental Setup

The simulated CPU frequency is the same as the HiFive SiFive Unleashed, running at 1GHz. Likewise, the simulated system uses 8GB of 2400MHz DDR4 RAM. The L1 and L2 caches are implemented using the classical cache models available in gXR5. They are 32KB, 8-way associative, and 2MB, 16-way associative, respectively. The software stack of both the simulated model and the HiFive Unleashed SoC includes the OpenSBI bootloader, the Linux kernel v5.8, and a 24GB buildroot filesystem.

[2] https://riscv.org/technical/specifications/

Results and Discussion

Five of the SPEC CPU2017 integer-rate benchmarks (out of 10 total applications) were successfully run on the validated simulator. The applications with a similar op-mix to stress-ng CPU class stressors have a mere 3.4% (500.*deepsjeng_r*) and 0.17% (541.*leela_r*) error in execution time. Figure 3 compares the execution time of SPEC CPU2017 applications running on the HiFive Unleashed SoC and on gXR5. A relatively large error in execution time for other applications is expected as they are much more memory intensive (i.e., they have nearly 40% Load/Store instructions), given only the MinorCPU model was calibrated. Calibration reduces the overall error in execution time from **44.3%** to **23.9%**.

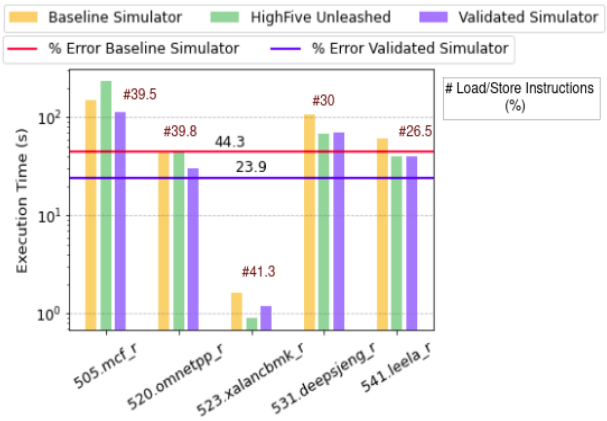


Figure 3: *Performance of Simulator vs actual hardware for SPEC CPU2017 suite*

This work has been partially supported by the EC H2020 WiPLASH project (GA No. 863337), the EC H2020 FVLLMONTI project (GA No. 101016776), and the ECO4AI project from HES-SO.

References

[1] *gXR5: A gem5-based full-system RISC-V simulator*. URL: https://www.epfl.ch/labs/esl/research/2d-3d-system-on-chip/gXR5.

[2] N. Binkert et al. "The gem5 simulator". In: *ACM SIGARCH computer architecture news* 39.2 (2011), pp. 1–7.

[3] Rajagopalan Desikan et al. "Measuring experimental error in microprocessor simulation". In: *ISCA* (2001).

[4] Quentin Huppert et al. "Memory hierarchy calibration based on real hardware in-order cores for accurate simulation". In: *DATE* (2021).

[5] Ayaz Akram and Lina Sawalha. "Validation of the GEM5 simulator for x86 architectures". In: *IEEE PMBS* (2019).

[6] Marco A Z Alves et al. "SiNUCA: A Validated Micro-Architecture Simulator". In: *2015 IEEE ICHPCC, IEEE ISCSS, and IEEE ICESS* (2015), pp. 605–610.

Validating Full-System RISC-V Simulator: A Systematic Approach

Karan Pathak, Joshua Klein, Giovanni Ansaloni, Marina Zapater and David Atienza

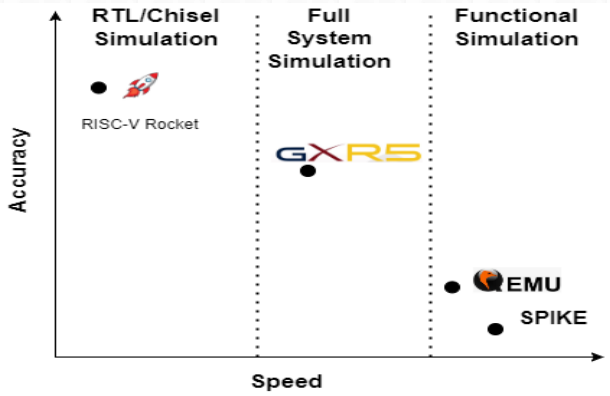
Overview

Simulate impact of architectural innovations quickly

Execute Real user's workload atop filesystem, kernel.

Motivation

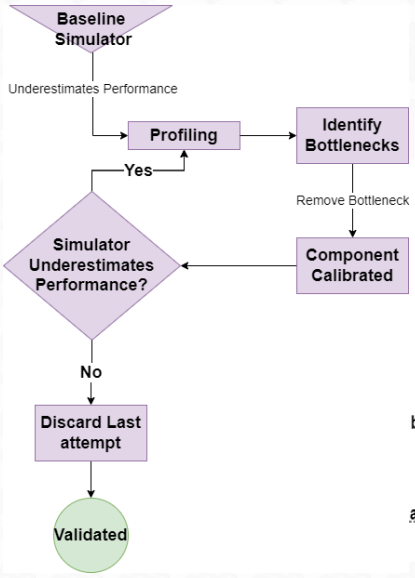
gXR5: gem5 e**X**tended for **R**ISC-**V**

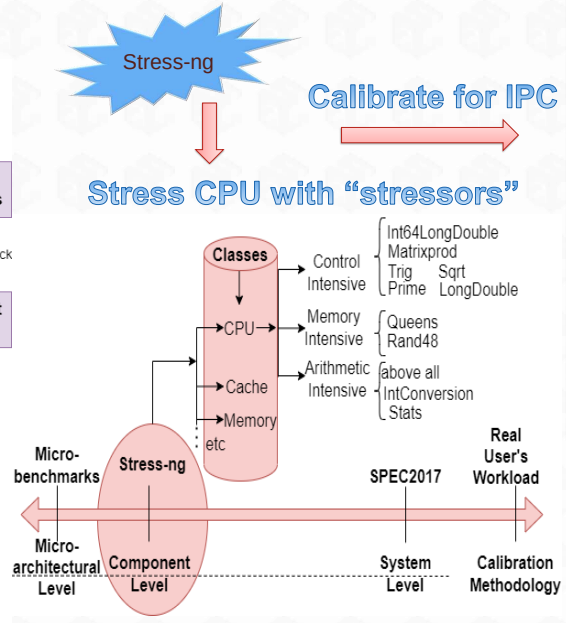


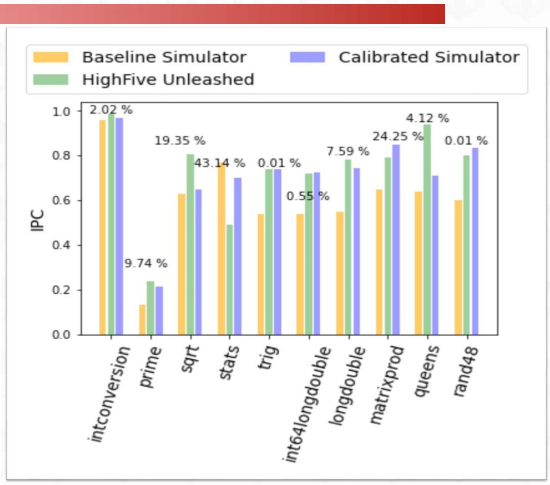
Contributions

- Enriches Simulator Validation Methodologies
- Full System RISC-V simulator validated against actual hardware
- Open Sourcing:
  - gXR5
  - bootloader
  - kernel
  - filesystem

Methodology







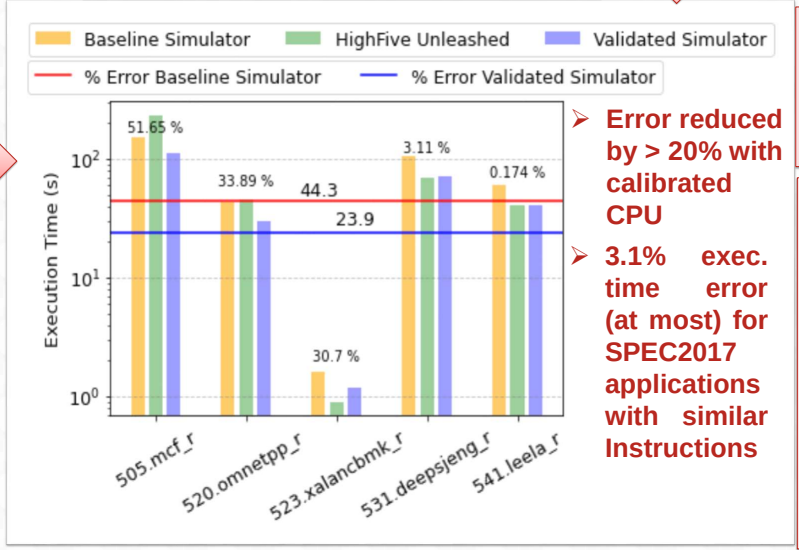
Simulator vs Hardware performance for stress-ng benchmarks

Validated simulator

Results

Component: Attribute	Baseline Value	Calibrated Value
Read Memory Functional Unit : Op Latency	4	2
Integer Division Functional Unit : Op Latency	33	19
Branch Predictor : Type	Tournament	Multiperspective Perceptron
Fetch2-Fetch1 : Backward Delay	1	0
Execute Unit : Branch Delay	1	3
L1 Cache : Associativity	2	8
L1 -Data Cache : Clusivity	Mostly exclusive	Mostly inclusive

Simulator vs Hardware performance for SPEC2017 applications



- Error reduced by > 20% with calibrated CPU
- 3.1% exec. time error (at most) for SPEC2017 applications with similar Instructions

Acknowledgments

This work has been partially supported by the EC H2020 WiPLASH project (GA No. 863337), the EC H2020 FVLLMONTI project (GA No. 101016776), and the ECO4AI project from HES-SO.

References

1. gXR5: A gem5-based full-system RISC-V simulator. url: https : / / www . epfl . ch / labs / esl / research / 2d - 3d - system-on-chip/gXR5.
2. N. Binkert et al. "The gem5 simulator". In: ACM SIGARCH computer architecture news 39.2 (2011), pp. 1–7.
3. Rajagopalan Desikan et al. "Measuring experimental error in microprocessor simulation". In: ISCA (2001).
4. Quentin Huppert et al. "Memory hierarchy calibration based on real hardware in-order cores for accurate simulation". In: DATE (2021).

A Validated Linux-capable RISC-V Simulator

Karan Pathak[†‡], Joshua Klein[†], Giovanni Ansaloni[†], Marina Zapater[†,*],
Georgi Gaydadjiev[‡] and David Atienza[†]
[†]*Embedded Systems Lab (ESL), École Polytechnique Fédérale de Lausanne*, Switzerland
[‡] *Faculty of Electrical Engineering, Mathematics and Computer Science, TU Delft*, The Netherlands
[*] *REDS. School of Engineering & Management Vaud, HES-SO University of Applied Sciences & Arts Western Switzerland*
Mail IDs(**To be ommitted for blind review**)

***Abstract*—The RISC-V ecosystem has been witnessing tremendous push by the industrial players to accelerate adoption of RISC-V chips in IoT, mobile, data centres, and embedded domains. However, there has yet been a publicly released RISC-V full-system simulator that has been validated against an actual hardware. We bridge this gap by presenting a first Linux capable, RISC-V full-system simulator that has been validated against an IP-protected commercial board as well as open source RISC-V system emulated on an FPGA. This work proposes a performance validation methodology, namely "component-level" calibration that is fast, accurate, and generic. The validated simulator error is brought down from 45% and 54% to 22.9% and 18.9% for selected SPEC2017 benchmarks applications, by calibrating the CPU model using the aforementioned methodology. The methodology also streamlines CPU performance validation of the simulators.**

***Index Terms*—Architectural simulator, performance validation, full-system (FS) simulator, in-order CPU, FPGA emulation, Hardware Performance Counters.**

I. Introduction

Computer Architects are often tasked with balancing the performance, power, and area requirements of the chip. The challenge is made stringent and exasperated by Time-to-Market constraints. Because hardware prototyping is both expensive and time consuming, architectural simulators come to the rescue by providing means to gauge the impact of architectural innovations quickly. Arguably, these simulators can avoid processor/product recall due to the systems failing to meet performance or power requirements. A full-system (FS) simulator expands the utility of such simulators by facilitating user space applications (that require OS libraries and a kernel) to be run on the simulated hardware rather quickly. Contrary to the application specific or user-mode clock driven simulations (e.g. RTL) that have poor simulation throughput on the order of Kilo Instruction per Second (KIPS) [1], the event-driven FS-simulator has higher simulation throughput (on the order of MIPS) [2] and far greater reconfigurability. The most important aspect of simulators is accuracy with which the hardware is modeled. Apart from being functionally correct, the simulators should also be representative of the performance of the hardware. Hence, validating simulator against hardware is essential to enhance the utility of the simulator.

This work has been partially supported by the EC H2020 WiPLASH project (GA No. 863337), the EC H2020 FVLLMONTI project (GA No. 101016776), and the ECO4AI project from HES-SO. The research stay was supported by Justus and Loiuse van Effen Research Grant.

gem5 e**x**tensions for **RISC-V** (gXR5 from now on) is a Linux capable fs-simulator built on top of open source gem-5 [3] architectural simulator. gem5 is an event-driven architectural simulator built by the merger of m5 and GEMS simulation frameworks. It provides ISA independent and tunable (micro) architectural models that allows simulation of a variety of ISAs (such as ARM and x86). It is highly reconfigurable by the inclusion of python (configuration) scripts that tune specific hardware attributes (e.g., latencies, bus widths, buffer sizes), while maintaining host performance through the use of compiled C++ code for all tuned modules.

A. Contributions

Architectural simulators suffer from three types of validation errors: Abstraction, Modeling, and Specification errors [4]. Abstraction errors are inherent to simulator and depend upon the chosen level of abstraction. It is a designer's choice. Modeling errors originate from the inability of the designer to capture the micro-architectural details of the actual hardware, usually due to generic models designed for greater reconfigurability. A functional validation removes the modeling errors. Specification errors are the most challenging to address as they arise when the designer is not aware of the micro-architectural details of the actual hardware, often because of proprietary rights (e.g. because the model is a black box). The proposed methodology of "component-level" calibration is an attempt to overcome these specification errors. The methodology is accurate, faster (in terms of simulation time), systematic (unlike existing methodology) and generic (that can be extended to other simulators to simulate various ISAs). Rest, our contributions are as follows:

- We implement a Linux-capable full system model for the RISC-V 64-bit general purpose architecture in gem5.
- We calibrate this model against a single core of the Linux-capable Sifive HiFive Unleashed SoC and an uni-core Rocket Chip emulated on VC707 Xilinx FPGA using our aforementioned methodology.
- We implement a new Gshare branch predictor model that is compatible with existing multi-threaded CPU models in gem5.
- Using selected SPEC CPU2017 benchmarks, we validate the performance statistics of our simulator to within a small margin of error with respect to the HiFive Unleashed SoC and Rocket Chip emulated on FPGA. This

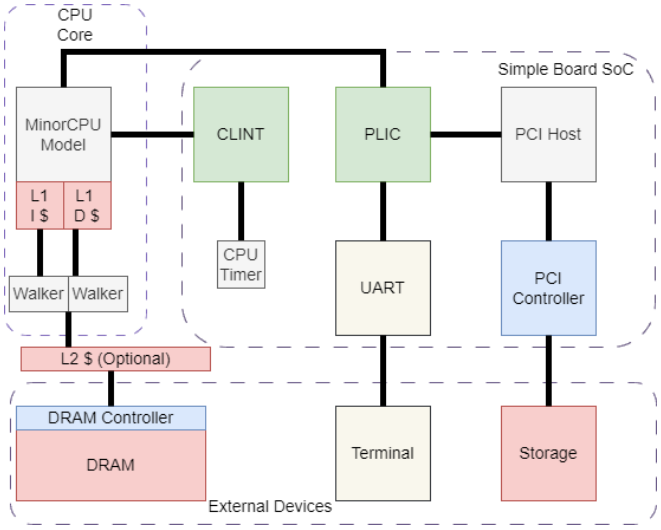


Fig. 1. Simulated Full System Model

removes the need for licensed FPGA emulation to run user's workload on Rocket Chip, making the Rocket Chip ecosystem truly open source.

- We release and open source our simulator and all supporting materials, including bootloader, kernel, filesystem, and technical manual, for quick and easy adoption by the RISC-V community.

B. Preliminary Work

The diagram of the adopted gXR5 full system model, which simulates execution on the uni-core HiFive Unleashed SoC and Rocket system emulated on VC707 FPGA, can be seen in Figure 1. The model takes advantage of the modularity of gem5 by integrating standard, included components of gem5. It uses the in-order MinorCPU model to simulate the in-order uni-core of the two aforementioned hardware targets. The core-local interruptor (CLINT) is modeled as a single simulation object with a timer for each CPU core. The rest of the SoC derives from the SimpleBoard [16]. The platform-level interrupt controller (PLIC) modeled is based on the one from FU540-C000 core, and is responsible for interrupts from external devices. The software stack of the gXR5 full-system simulator includes an OpenSBI bootloader, the Linux v5.8 kernel, and a buildroot file system. Since we base our validation efforts on a single-core system, the in-order Minor CPU model is discussed in detail.

The Minor CPU derives from the BaseCPU model that implements the basic functionalities such as setting up a fetch request, handling pre-execute setup, handling post-execute actions, advancing the Program Counter, etc. It is a four-stage pipelined CPU model with following stages: Fetch1, Fetch2, Decode, and Execute. These stages are connected through input (size-tunable) buffers that hold the instructions in case of a stall. The Fetch1 stage fetches the instructions from the L1 Instruction cache, while Fetch2 stage has a branch predictor unit with Branch History Table (BHT) and Branch Target Buffer (BTB). The Decode stage converts the instruction into micro-operations before passing them to the Execute stage. The Execute stage hosts the arithmetic functional units and the Load/Store unit (LSU). The functional units are modeled as black boxes with a finite *Operation (Op) Latency*.

TABLE I
TECHNICAL SPECIFICATIONS OF SIMULATED MODELS AND TARGET HARDWARE.

Component	**HiFive Unleashed**		**Rocket**	
	Hardware	gXR5	Hardware	gXR5
CPU Core	U54	MinorCPU	Big Core	MinorCPU
CPU	RV64GC		RV64G	RV64GC
CPU Freq	1		0.1 GHz	1.067 GHz
L1 I & D $	32KB 8-Way		16KB 4-Way	
L2 $	2 MB 16-Way		None	
MMU	Sv39			
Modes	Machine, Supervisor, User			
RAM	DDR4	DDR4_4x16	DDR3	DDR3_8x8
RAM Freq	2400 MHz		200 MHz	2133 MHz
RAM Size	8GB		4GB	
System Bus	TileLink	XBar	TileLink	XBar

C. Target Hardware

The Sifive Unleashed was the first Linux-capable RISC-V system that was commercially available. The soundness of the proposed methodology to validate is illustrated by validating gXR5 against the IP protected processor, details of whose micro-architecture are not open source. On the other hand, the Rocket core is one of the most popular open source RISC-V hardware systems [5]. It has been chosen as a target hardware to prove the fidelity of the proposed methodology. The micro-architecture of the two target CPU cores is described below:

- **U54 core** is an in-order 5-stage pipelined CPU with 32KB 16-way L1 Instruction and Data caches. It has a branch predictor unit with a 30-entry BTB that caches the target of taken branches, a 256-entry BHT that stores the direction of conditional branches, and a 6-entry return-address stack (RAS). The latency of the integer multiplier is 5 clock cycles. The integer division unit has a latency between 2 and 65 clock cycles [**?**]. The details of rest of the micro-architecture is closed-source and therefore poses a challenge as it a source of specification errors.
- **Rocket Core** is a 5-stage in-order scalar processor with L1 Instruction and Data caches. The default configuration of the core includes floating point units (Single Precision and Double Precision Fused Multiply Accumulate and Division). The integer functional units include an 8-cycle iterative integer multiplier (with one cycle each to load the operands and place the result on the output bus). It comes with an integer ripple carry adder and integer division unit. It has a Gshare branch predictor that uses hash of branch address XORed with hash of global history to index into Pattern History Tables (PHTs)

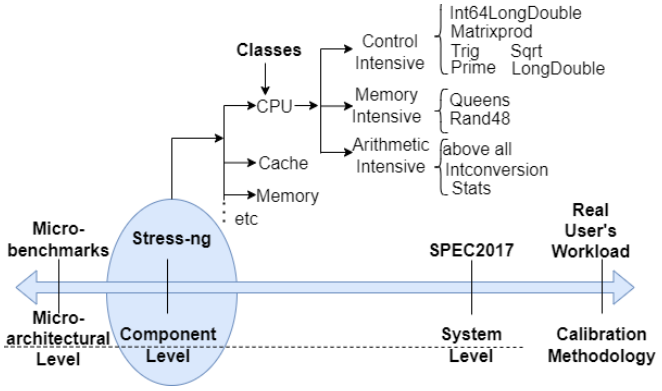


Fig. 2. Methodology: Component Level Calibration

containing a 2-bit counter. The hash functions are as follows:

$$Hash\_1(gh) = \sqrt{\frac{3}{2}} * 2^{History\_length} * gh \quad (1)$$

$$Hash\_2(pc) = pc >> log_2(fetch\ bytes) \quad (2)$$

The hashed values are resized before matching the pattern. The L1 Instruction cache is three stage pipeline with data access latency of 2 clock cycles under a hit. Similarly, the L1 Data cache completes the tag comparison and serves the request of data fetch in 2 clock cycles in case of a tag hit. There exists an additional stage for hosting the Miss Status Holding Register (MSHR) that keeps a track of hits under a miss. The Table I summarises the simulator set-up and the corresponding hardware specifications.

The attributes of the simulator such as latency of modeled functional units, cache access latency, etc., are matched with those of the hardware (subject to open-sourced technical specifications) . We call this configuration of the simulator as the 'Baseline' simulator.

II. Related Works

One of the earliest works on performance validation by Desikan et al. [7] achieved 18% simulator error in IPC for macro-benchmarks derived from the SPEC CPU2000 benchmark suite. The methodology can be best described as 'validation by inspection', as multiple execution of the macro-benchmarks in SimAlpha simulator and the hardware are required to bridge the performance (IPC) gap. More recent attempts include Butko et al. [8], which validated the gem5 simulator for ARM A8 (dual core) and A9 cores executing PARSEC benchmarks with the simsmall input set. Similar work has been carried out by Qureshi et al. [9], which obtained a 4% simulator error compared to the ARM JUNO platform in terms of execution time for a real-time video transcoding app. However, considerable time (often a few days) is required to execute a single instance of the macro-benchmark in the simulator. In order to expedite the validation process, an alternate approach used synthetic micro-benchmarks (as Training set) to calibrate the region of interest prior to executing the macro-benchmarks

TABLE II
Past simulator validation efforts with reported errors for benchmarks.

Validated Simulator	Target Hardware	Training Set Error	Test Set Error
SimAlpha	DS-10L workstation (Compaq Alpha 21264 processor)	Synthetic microbenchmarks IPC Error of less than 2 %	selected macrobenchmarks derived from SPEC2000 suite IPC Error of 18 [7] %
SiNUCA	Sandy bridge processor	Synthetic microbenchmarks IPC Error of 9 %	SPEC2006 suite having an IPC Error of 19% [12]
gem5	ARM Versatile Express TC2 development board	SPEC and PARSEC	SPEC and PARSEC Run Time Error of 13 % and 11 % [4]
gem5	Cortex-A53 core of MediaTek Helio X20 SoC	Synthetic microbenchmarks	SPEC2006 suite IPC Error of 20 % [11]
gem5	Arm R8 CPU	Embench workload CPI Error of 13 %	No macrobenchmarks were run [13]
gem5	Intel Core-i7 (Haswell Microarchitecture)	Synthetic microbenchmarks IPC Error of 6 %	No macrobenchmarks were run [10]
gem5	ARM Cortex-A8 (dual core, Snowball SDK), ARM Cortex-A9 (single core,BeagleBoard-xM SDK)	PARSEC	Selected PARSEC benchmarks (with simsmall input set), Run time Error of 8 % for both [14]
GEMS	ARM Cortex-A9 (Snowball SKY-S9500-ULP-C01)	SPLASH-2, ALPBench, STREAM	same as test set, Run time Error of 1.39% to 17.94 % [8]

(as Test set) such as SPEC/PARSEC. We call this methodology "micro-architectural level calibration".

The Table II summarises the validation efforts and the methodology used. Despite past efforts, the design of the micro-benchmarks and the validation methodology has not been streamlined. Attempts have been made to use the Pearson's correlation between micro-architectural events and error in IPC [10] to streamline the calibration methodology. However, the validated simulator has not been tested on macro-benchmarks representative of user's workload. Huppert et al. [11] provide methodology for memory hierarchy calibration achieving a mean error of 20% in IPC for the SPEC CPU2006 benchmark suite. This simplistic methodology (of 'data pinning') can not be extended to other components such as the CPU, Branch Predictors, TLBs, and Page Walkers.

III. Methodology

We propose a "component-level calibration" methodology for validating simulators. Figure 2 situates the proposed methodology among the existing ones.The proposed methodology uses stress-ng benchmarks [6] for fine-tuning the CPU model in gem5.

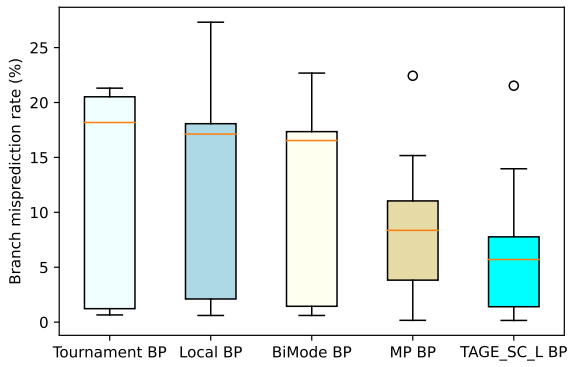


Fig. 3. Mis-prediction rates of branch predictors executing the stress-ng benchmark.

A. Stress-ng benchmarks

The stress-ng benchmarks were originally designed to perform accelerated stress-test of a particular component of the computing system and cause thermal overruns. The benchmark suite has been divided into classes of "stressors", with each class stressing a particular component and often containing tens to over 100 stressors. The CPU class stressors number over 100 at time of writing, and thus we choose a representative subset of 10 stressors using Principal Component Analysis (PCA) to ensure diversity of the type of workload. Furthermore, we reduce the need for extensive characterisation of the hardware using Hardware Performance Counters as only Instruction and Cycle count are being used (IPC). Additionally, the stress-ng micro-benchmarks are much more representative of the time-complexity of real-user's workload than the synthetic microbenchmarks [6]. As a result, the validated simulators have less performance disparity when macro-benchmarks (SPEC suites) are executed. The following section profiles and classifies the selected CPU class stressors. The classification is in line with existing validation efforts [7].

B. Profiling and Classification

The selected CPU class stressors were profiled using gXR5 with the baseline configuration. They are classified into three categories:

- **Control Intensive:** having complex control structures (e.g. nested for-if-else) with at least 100 branches per 1000 instructions.
- **Memory Intensive:** having 20% or higher load/store instructions in the total op-code mix of the program. A maximum 30% instructions are load/store (for stressors "queens" and "rand48").
- **Arithmetic Intensive:** having 20% or more Instructions using Integer/Float functional units. All the stressors are arithmetic/compute intensive.

The stress-ng benchmarks are used as a 'test set' to fine-tune the attributes of gXR5 targeting Instructions per Cycle

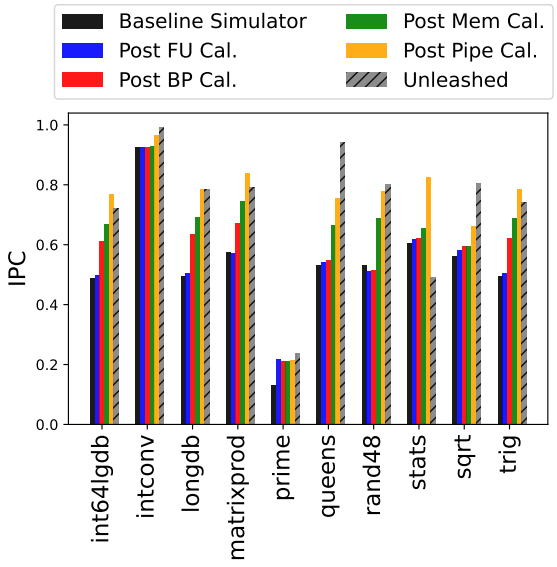


Fig. 4. Performance of gXR5 and Unleashed at various stages of calibration.

(IPC) to reduce performance disparity. IPC has been chosen as the Figure of merit since it captures minuscule changes in performance of both simulated system and hardware. Once, calibrated, the SPEC suite is executed on the 'validated' simulator.

IV. Results and Discussion

A. Calibrating for the U54 Core

The Baseline simulator underestimates the performance (IPC) compared to the actual hardware for all the stressors (Figure 4). The bottlenecks in the simulated system were identified and removed to improve the simulated system's performance. The fine-tuning led to decreasing the latency of functional units (such as Integer Division and Add). After Functional Unit calibration, the bottleneck in systems performance were control hazards. Hence, a design space exploration of the existing branch predictors was performed to reduce the branch miss-predictions and associated CPU stalls (Figure 3). The Multiperspective Perceptron branch predictor model gives least Mean Absolute Percentage Error (MAPE) in IPC by improving the simulated system performance. The TAGE_SC_L (TAgged GEometric length with statistical correlator and loop predictor) branch predictor model has larger outliers for some of the stressors that gives higher MAPE in IPC. The branch mis-predictions cause thrashing in the L1 instruction and data caches. Hence, L1 cache and the load/store units were tuned after switching to Multiperspective Perceptron branch predictor (MPBP). The memory related stalls were removed by decreasing the data access latency of L1 cache and making L2 cache inclusive of L1 cache. This significantly improves performance of stressor having working set size greater than L1 cache capacity. Finally, the input buffers to execute unit were made larger to ensure higher functional unit utilisation

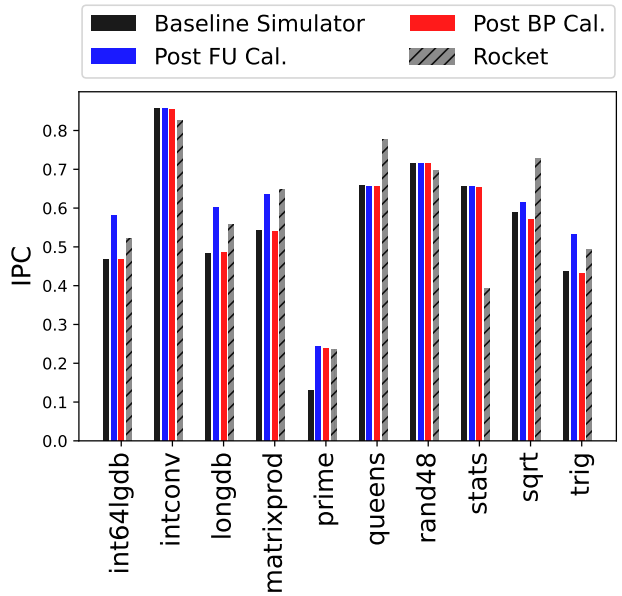


Fig. 5. Performance of gXR5 and Rocket Chip at various stages of calibration

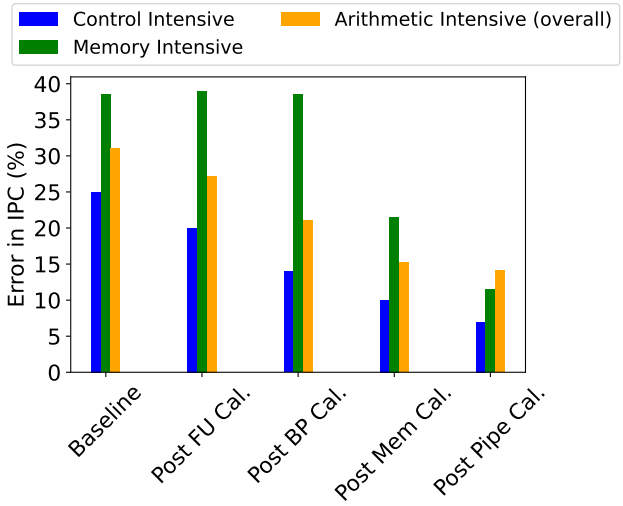


Fig. 6. MAPE in IPC for stressors run on gXR5 and Sifive Unleashed at various calibration stages

even in case of control-related stalls. The IPC at various stages of calibration is depicted in Figure 4.

B. Calibrating against Rocket core

The rocket system was emulated on the VC707 Xilinx FPGA using the Vivado High-Level Synthesis (HLS) tool. The synthesiser makes use of the DSP48E1 slices to implement the floating point units (SPFMA, DPFMA and FPDiv) in order to maximise the performance of the system. The DSP48E1 slice has a single latency (25 x 18 bits) multiplier with shift logic at the output bus to facilitate larger operand multiplications. The output of the multiplier is fed to a single cycle adder/subtractor via a register. The input ports have registers that can break the critical path by absorbing operands into registers (thereby increasing latency of operation). Hence, the latency of these functional units were fixed using the same selected stress-ng stressors, targeting reduction in IPC. Both single and double precision floating point instructions are mapped to a single 'op-class' inside gem5, which in-turn is mapped to *FloatMultAcc* functional unit. A heuristic approach or ILP modeling can give minimum possible latency of synthesised functional units using DPS48E1 slice that serve as the starting point of DSE for fixing *Op Latency* of the Floating Point units. The *Op Latency* of FloatMultAcc and FloatDiv is fixed at 6 clock cycles and 9 clock cycles respectively, as it gives least MAPE in IPC for stress-ng benchmarks.

Finally, a Ghsare branch predictor [15] model is implemented in gXR5/gem5. This model is compatible with gem5 CPU (multi-threaded) models and can be configured via the corresponding python scripts in gem5. The IPC of stressors in simulated system and hardware is depicted in Figure 5.The implementation of gshare branch predictor model reduces the MAPE in control intensive benchmarks from 15% to less than 8 %. The MAPE in IPC for stress-ng benchmarks has been depicted in Figure 7.

The final configuration of the gXR5 calibrated for Sifive Unleashed and Rocket Chip is called the 'validated simulator'. The stress-ng benchmarks achieve the component-level validation. But validation of the entire simulated system necessitates executing workload representative of user-space such as SPEC benchmark suite. SPEC2017 int rate applications (with test inputs) are run on the simulator and the target hardware. The execution time of the application gives an estimate of performance of the simulated system. Hence, the execution time is chosen as the Figure of merit to test the validated simulator while executing SPEC suite applications. However, this is possible only if the simulator and hardware operate at the same frequency. Otherwise, clock cycles elapsed while executing applications is chosen as an alternative figure of merit.

C. Validated Simulator against Sifive Unleashed

The run time of the SPEC applications are compared for the baseline simulator, validated simulator, and the hardware. Figure 8 depicts baseline error of 44% and the final (validated simulator) error of 23.9%. The specification error is brought down by more than 20% just by calibrating the MinorCPU model in gXR5. As expected, the three application having higher load/store instructions (mcf, omnetpp, and xalancbmk) have higher run time error. For the other two applications, the run time error is mere 3.4% and 1.7%, highlighting superior results of the proposed methodology.

D. Validated Simulator against Rocket System

The Rocket system emulated on VC707 FPGA runs at a frequency different than the system modeled in gXR5.

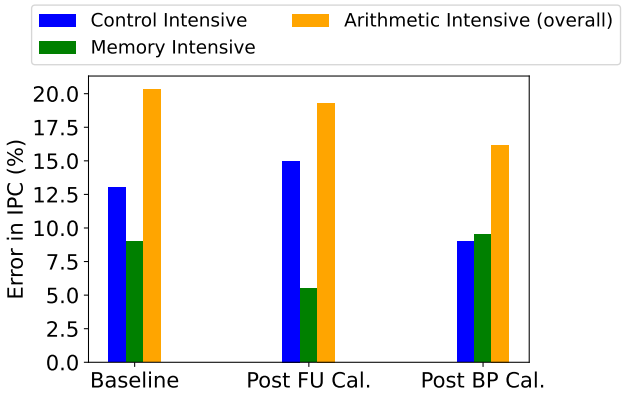


Fig. 7. MAPE in IPC for stressors run on gXR5 and Rocket Chip at various calibration stages

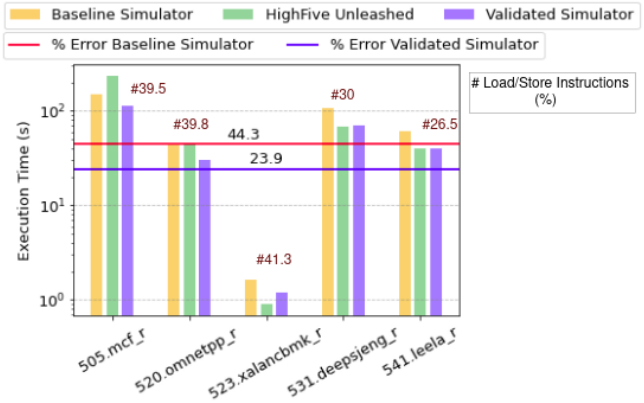


Fig. 8. Performance of gXR5 Vs. Sifive Unleashed for selected SPEC2017 suite applications.

Moreover, the ratio of DRAM Controller frequency and CPU operating frequency is kept at 10.665:1 for both Hardware and gXR5. The number of clock cycles elapsed while executing SPEC suite applications is used to compare the performance statistics of the hardware and gXR5. The MAPE in execution cycles is brought down to 18.9 % (Figure 9), showcasing the fidelity of the proposed methodology for performance validation.

V. Conclusion and Future Work

A single run of simulation for stress-ng benchmarks takes less than 180 minutes to execute stressors for 5 seconds. No more than 15 simulation runs were required to calibrate the MinorCPU model. Many of these simulations were run in parallel, thereby reducing the validation effort time from days to hours. Moreover, the only Hardware Performance Counts (HPCs) used were instructions and clock cycles (for calculating IPC), thereby significantly reducing the effort spent in characterizing the hardware. We achieve an error of 22.9% and 18.9 % for the two target hardware, just by fine-tuning MinorCPU model. The methodology can be further extended

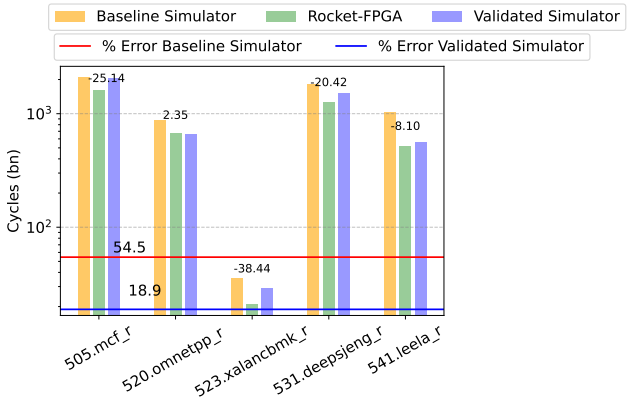


Fig. 9. Performance of gXR5 Vs. Rocket Chip for selected SPEC2017 suite applications.

for other component calibration, especially memory hierarchy. Moreover, a good ground to test the methodology would be to simulate multi-core systems with shared memory. Ruby caches in gem5 provide a much detailed micro-architecture including various cache coherency fabrics. The stress-ng stressors can be run on multiple cores to tune the modeled (shared) memory, Interconnects, etc.

References

[1] López-Paradís et al., "Fast Behavioural RTL Simulation of 10B Transistor SoC Designs with Metro-Mpi," In DATE 2023, Antwerp, Belgium.

[2] N. Zurstraßen et al., "par-gem5: Parallelizing gem5's Atomic Mode," In DATE 2023, Antwerp, Belgium.

[3] N. Binkert et al.,"The GEM5 simulator," In ACM SIGARCH Comput. Archit. News, 2011.

[4] A. Gutierrez et al., "Sources of error in full-system simulation," In ISCA 2014.

[5] A. Dörflinger, "A comparative survey of open-source application-class RISC-V processor implementations," In ACM ICCF 2021.

[6] "Stress-ng." ubuntu-wiki. Accessed January 01,2023. [Online]. Available: https://wiki.ubuntu.com/Kernel/Reference/stress-ng

[7] R. Desikan, D. Burger and S.W. Keckler, "Measuring experimental error in microprocessor simulation," In ISCA 2001.

[8] A. Butko, R. Garibotti, L. Ost and G. Sassatelli, "Accuracy evaluation of GEM5 simulator system," In International Workshop on ReCoSoC, 2012.

[9] Y. M. Qureshi et al., "Gem5-X: A Gem5-Based System Level Simulation Framework to Optimize Many-Core Platforms," In SpringSim 2019.

[10] A. Akram and L. Sawalha, "Validation of the gem5 Simulator for x86 Architectures," In IEEE/ACM PMBS 2019.

[11] Q. Huppert et al., "Memory Hierarchy Calibration Based on Real Hardware In-order Cores for Accurate Simulation," In DATE 2021.

[12] M. A. Z. Alves et al., "SiNUCA: A Validated Micro-Architecture Simulator," In 2015 IEEE ICHPCC, IEEE ISCSS, and IEEE ICESS.

[13] I. Wang, P. Chakraborty, Zi Yu Xue, and Yen Fu Lin. 2022. Evaluation of gem5 for performance modeling of ARM Cortex-R based embedded SoCs. In Microprocess. Microsyst. 2022.

[14] F. A. Endo, D. Couroussé and H. -P. Charles, "Micro-architectural simulation of in-order and out-of-order ARM microprocessors with gem5," In SAMOS XIV, 2014.

[15] S. McFarling, "Combining Branch Predictors," Technical Report TN-36, Digital Western Research Lab., June 1993.

[16] Robert Scheffel, "Simulation of RISC-V based Systems in gem5," Research Master thesis, TU Dresden, 2018.

[17] "SiFive FU540-C000 Manual v1p4." Accessed January 01,2023. [Online]. Available:https://sifive.cdn.prismic.io/sifive/d3ed5cd0-6e74-46b2-a12d-72b06706513e_fu540-c000-manual-v1p4.pdf

B

RISC-V ISA selected Instructions

Abbreviation	Instruction (Description)
LW	Load Word
LH	Load Half Word
LHU	Load Half Word Unsigned
LB	Load Byte
LBU	Load Byte Unsigned
MUL	Multiply 64-bit numbers and places the lower 64 bits in the destination register
MULH	(signed)64-bit×(signed)64-bit multiplication and places the upper 64-bits in the destination register
MULHU	(unsigned)64-bit×(unsigned)64-bit multiplication and places the upper 64-bits in the destination register
DIV	Division of 64-bit by 64-bit number
DIVU	Division of unsigned 64-bit number by unsigned 64-bit number
LD	Load DoubleWord
ANDI	Logical AND with Immediate value
ADD	Add the operands in the register
SRAI	Shift Right Arithmetic Immediate
SLLI	Shift Right Logical Immediate
BNE	Branch if Not Equal
BEQ	Branch if Equal

Table B.1: Selected RISC-V Instructions

C

C Code

C.1. ILP Modeling: Parsing and Generating constraint equations

Listing C.1: Parsing the sequncing table

```
void read_header( std::string const& line, uint32_t& m, uint32_t& n )
  {
    std::string str;
    std::stringstream ss( line );
    std::getline( ss, str, '□' );
    m = std::stoi( str );

    std::getline( ss, str, '□' );
    n = std::stoi( str );

    std::getline( ss, str, '□' );
    latency_bound = std::stoi( str );
  }

  void add_NOP_type()
  {
    resources['0'] = Resource( {1, 1} );
  }

  void add_resource_type( std::string const& line )
  {
    std::string str;
    std::stringstream ss( line );

    std::getline( ss, str, '□' );
```

```
  char const name = str[0];

  std::getline( ss, str, '□' );
  uint32_t const delay = std::stoi( str );

  std::getline( ss, str, '□' );
  uint32_t const num = std::stoi( str );

  assert( resources.find( name ) == resources.end() );
  resources[name] = Resource( {delay, num} );
}

void add_first_NOP()
{
  operations.emplace_back( Operation( {'0', {}} ) );
}

void add_operation( std::string const& line )
{
  std::string str;
  std::stringstream ss( line );

  std::getline( ss, str, '□' );
  char const type = str[0];
  assert( resources.find( type ) != resources.end() );

  operations.emplace_back( Operation( {type, {}} ) );
  auto& pred = operations.back().predecessors;

  while ( std::getline( ss, str, '□' ) )
  {
    pred.emplace_back( std::stoi( str ) );
  }
}
```

Listing C.2: Generating ILP equations

```
void operations_starts_once()
{
  for ( auto i = 1u; i <= num_operations; ++i )
  {
    Constraint constraint;
    constraint.type = EQ;
    constraint.constant = 1;
    for ( auto l = 1u; l <= num_timeframes; ++l )
    {
```

```
        constraint.variables.emplace_back( Variable( {i, l} ) );
        constraint.coefficients.emplace_back( 1 );
      }
      constraints.emplace_back( constraint );
    }
  }
void sequencing_relations()
  {

    for ( auto i = 1u; i <= num_operations; ++i )
    {
      auto predec = prob.operations.at( i ).predecessors;
       if (!predec.empty()){
        for ( auto p : predec){
          Constraint constraint;
          constraint.type = GE;
          char resourcetype = prob.operations.at( p ).type;
          //printf("resource type is : %d", resourcetype);
           constraint.constant = prob.resources.at(resourcetype).delay;
          //printf("delay is : %d\n", res);

          for ( auto l = 1u; l <= num_timeframes; ++l ){
            constraint.variables.emplace_back( Variable( {i, l} ) );
            constraint.coefficients.emplace_back( l );
            //printf("var and coeff are %d and %d \n", i, l);
            if (p ==0) continue;
            constraint.variables.emplace_back( Variable( {p, l} ) );
            constraint.coefficients.emplace_back( -signed(l) );
            // printf("predecessors of %d is %d \n", i, p);
            // printf("coeff is: %d\n", -l);
          }
          constraints.emplace_back( constraint );
        }
      }
      else{
        Constraint constraint;
        constraint.type = GE;
        constraint.constant = 0;//here dj coressponding
        for ( auto l = 1u; l <= num_timeframes; ++l ){
          constraint.variables.emplace_back( Variable( {i, l} ) );
          constraint.coefficients.emplace_back( l );
        }
        constraints.emplace_back( constraint );
      }
```

```
  }

}

void resource_bounds()
{

  //auto it = prob.resources.begin();
  for ( auto it = prob.resources.begin(); it != prob.resources.end(); ++it ){
    // printf("resource types is %d\n", it->first);
    if (it->first == 48) continue;
    int max_resource = it->second.num;
    int delay_resource = it->second.delay;
    for( auto l = 1u; l <= num_timeframes; ++l ){
      Constraint constraint;
      constraint.type = LE;
      constraint.constant = max_resource;
      auto m = l- delay_resource + 1u;
      if(m<1||m>=num_timeframes) continue;
      for(auto index = m; index<l+1u;index++){
        for ( auto i = 1u; i <= num_operations; ++i ){
          auto resourcetype = prob.operations.at( i ).type;
          if (resourcetype == it->first){
          constraint.variables.emplace_back( Variable( {i, index} ) );
          constraint.coefficients.emplace_back( 1 );
          }
         else
         continue;
         //printf("No operations of type found for this timeframe\n");
        }
      }
      //  printf("Equation complete! Moving on to next equation\n");
       constraints.emplace_back( constraint );
    }
  }
}

/*printing ILP and dumping to schedule.lp file*/
void print_lp( std::ostream& os = std::cout ) const
{
  /* the objective function */
  os << "min:";
  for ( auto l = 1u; l <= num_timeframes; ++l )
  {
    os << "+" << l << "x" << num_operations << "_" << l;
```

```
  }
  os << "□-1;" << std::endl;

  /* the constraints */
  for ( auto const& con : constraints )
  {
    assert( con.variables.size() == con.coefficients.size() );
    for ( auto v = 0u; v < con.variables.size(); ++v )
    {
      auto& var = con.variables.at( v );
      auto& cof = con.coefficients.at( v );
      assert( 1 <= var.i && var.i <= num_operations );
      assert( 1 <= var.l && var.l <= num_timeframes );
      if ( cof == 0 ) { continue; }
      if ( cof > 0 ) { os << "+"; }
      if ( cof > 1 || cof < -1 ) { os << cof; }
      else if ( cof == -1 ) { os << "-"; }
      os << "□x" << var.i << "_" << var.l << "□";
    }
    switch ( con.type )
    {
      case GE: { os << ">="; break; }
      case LE: { os << "<="; break; }
      case EQ: { os << "="; break; }
      default: assert( false );
    }
    os << "□" << con.constant << ";" << std::endl;
  }

  /* variable type declaration */
  os << "binary□";
  for ( auto i = 1u; i <= num_operations; ++i )
  {
    for ( auto l = 1u; l <= num_timeframes; ++l )
    {
      os << "x" << i << "_" << l;
      if ( i != num_operations || l != num_timeframes )
      {
        os << ",□";
      }
    }
  }
  os << ";" << std::endl;
}
```

C.2. GShare Brach Predictor model in gem5

The following code is for gem5 compatible Gshare branch predictor model used with MinorCPU mode.

Listing C.3: Gshare Branch Predictor

```
#include "cpu/pred/gshare.hh"
#include "base/bitfield.hh"
#include "base/intmath.hh"

GshareBP::GshareBP(const GshareBPParams *params)
    : BPredUnit(params),
      globalHistoryReg(params->numThreads, 0),
      globalHistoryBits(ceilLog2(params->globalPredictorSize)),
      globalPredictorSize(params->globalPredictorSize),
      globalCtrBits(params->globalCtrBits),
      globalCtrs(globalPredictorSize, SatCounter(globalCtrBits))
{
    if (!isPowerOf2(globalPredictorSize))
        fatal("Invalid global history predictor size.\n");

    globalHistoryMask = globalPredictorSize - 1;
    // state counter for FSM
    localThreshold  = (unsigned) (ULL(1) << (globalCtrBits  - 1)) - 1;
}

/*
For Unconditional Branch, it is always taken*/
void
GshareBP::uncondBranch(ThreadID tid, Addr pc, void * &bpHistory)
{
    BPHistory *history = new BPHistory;
    history->globalHistoryReg = globalHistoryReg[tid];
    history->finalPred = true;
    bpHistory = static_cast<void*>(history);
    updateGlobalHistReg(tid, true);
}

void
GshareBP::squash(ThreadID tid, void *bpHistory)
{
    BPHistory *history = static_cast<BPHistory*>(bpHistory);
    globalHistoryReg[tid] = history->globalHistoryReg;

    delete history;
```

```
}

/*
 * A hash of the global history register
 and a branch's PC is used to index into counter,
 which is used to select the final branch prediction.
 */
bool
GshareBP::lookup(ThreadID tid, Addr branchAddr, void * &bpHistory)
{
    unsigned globalHistoryIdx = (((branchAddr >> instShiftAmt)
                                ^ globalHistoryReg[tid])
                                & globalHistoryMask);

    assert(globalHistoryIdx < globalPredictorSize);
    bool final_prediction = globalCtrs[globalHistoryIdx]> localThreshold;
    BPHistory *history = new BPHistory;
    history->globalHistoryReg = globalHistoryReg[tid];
    history->finalPred = final_prediction;
    bpHistory = static_cast<void*>(history);
    updateGlobalHistReg(tid, final_prediction);

    return final_prediction;
}

void
GshareBP::btbUpdate(ThreadID tid, Addr branchAddr, void * &bpHistory)
{
    globalHistoryReg[tid] &= (historyRegisterMask & ~ULL(1));
}

void
GshareBP::update(ThreadID tid, Addr branchAddr, bool taken, void *bpHistory,
                 bool squashed, const StaticInstPtr & inst, Addr corrTarget)
{
    assert(bpHistory);

    BPHistory *history = static_cast<BPHistory*>(bpHistory);
    // taken is the actual judgement. The global history register
    // needs to be updated incase of squash and taken only
    // accordingly
    if (squashed) {
        if (taken)
            globalHistoryReg[tid] = (history->globalHistoryReg << 1) | taken;
        return;
```

```
    }

    unsigned globalHistoryIdx = (((branchAddr >> instShiftAmt)
                                ^ history->globalHistoryReg)
                                & globalHistoryMask);

    assert(globalHistoryIdx < globalPredictorSize);

    if (taken) {
        globalCtrs[globalHistoryIdx]++;
    } else {
        globalCtrs[globalHistoryIdx]--;
    }

    delete history;
}

void
GshareBP::updateGlobalHistReg(ThreadID tid, bool taken)
{
    globalHistoryReg[tid] = taken ? (globalHistoryReg[tid] << 1) | 1 :
                                (globalHistoryReg[tid] << 1);
    globalHistoryReg[tid] &= historyRegisterMask;
}

GshareBP*
GshareBPParams::create()
{
    return new GshareBP(this);
}
```

D

Stress-ng Benchmark suite

The description of the selected stress-ng benchmark of **"cpu" class** with methods is given below:

1. **Prime** : Find the first 10000 prime numbers using a slightly optimised brute force naïve trial division search.
2. **Sqrt** : Finding Square Root of Double and Long Double numbers less than 16384
3. **Queens** : Solving Queens Problem for sizes 1 to 12.
4. **Rand48** : 16384 iterations of drand48 and lrand48 where
   - The drand48 and erand48 functions return nonnegative double-precision floating-point values uniformly distributed over the interval $[0.0,\ 1.0)$.
   - The lrand48 and nrand48 functions return nonnegative long integers uniformly distributed over the interval $[0,\ 2^{31})$.
5. **Matrixprod** : Matrix product of two 128 × 128 matrices of double floats.
6. **Longdouble** : 1000 iterations of a mix of long double precision floating point operations.
7. **Stats** :Calculate minimum, maximum, arithmetic mean, geometric mean, harmonic mean and standard deviation on 250 randomly generated positive double precision values.
8. **Trig** : Compute $sin(\theta) \times cos(\theta) + sin(2\theta) + cos(3\theta)$ for float, double and long double sine and cosine functions where $\theta = 0$ to 2π in 1500 steps.
9. **Intconversion** : Perform 65536 iterations of integer conversions between int16, int32 and int64 variables.
10. **Int64longdouble** :1000 iterations of a mix of 64 bit integer and long double precision floating point operations.

The description of the selected stress-ng benchmark of **"memory" class** with methods is given below:

1. **memcpy N**: Start N workers that copy 2MB of data from a shared region to a buffer using memcpy(3) and then move the data in the buffer with memmove(3) with 3 different alignments. This will exercise processor cache and system memory.
2. **membarrier N**: Start N workers that exercise the membarrier system call (Linux only).
3. **memfd N**: Start N workers that create 256 allocations of 1024 pages using memfd_create(2) and ftruncate(2) for allocation and mmap(2) to map the allocation into the process address space. (Linux only).
4. **mlock N**: Start N workers that lock and unlock memory mapped pages using mlock(2), munlock(2), mlockall(2) and munlockall(2). This is achieved by the mapping of three contiguous pages and then locking the second page, hence ensuring non-contiguous pages are locked . This is then repeated until the maximum allowed mlocks or a maximum of 262144 mappings are made. Next, all future mappings are mlocked and the worker attempts to map 262144 pages, then all pages are munlocked and the pages are unmapped.
5. **mmap N**: Start N workers continuously calling mmap(2)/munmap(2). The initial mapping is a large chunk (size specified by –mmap-bytes) followed by pseudo-random 4K unmappings, then pseudo-random 4K mappings, and then linear 4K unmappings. Note that this can cause systems to trip the kernel OOM killer on Linux systems if not enough physical memory and swap is not available. The MAP_POPULATE option is used to populate pages into memory on systems that support this. By default, anonymous mappings are used, however, the –mmap-file and –mmap-async options allow one to perform file based mappings if desired.

E

SPEC2017 Benchmarks

The functionality implemented by the selected SPEC integer rate benchmark suite applications has been desrcibed below:

1. **502.gcc_r** : It is based on GCC Version 4.5.0. It generates code for an IA32 processor. The benchmark runs as a compiler with many of its optimization flags enabled.
2. **505.mcf_r** : It is a benchmark which is derived from MCF, a program used for single-depot vehicle scheduling in public mass transportation. The program is written in C. The benchmark version uses almost exclusively integer arithmetic.
3. **520.omnetpp_r** : The benchmark performs discrete event simulation of a large 10 gigabit Ethernet network. The simulation is based on the OMNeT++ discrete event simulation system, a generic and open simulation framework. OMNeT++'s primary application area is the simulation of communication networks, but its generic and flexible architecture allows for its use in other areas such as the simulation of IT systems, queueing networks, hardware architectures or business processes as well.
4. **541.leela_r** : It is a Go playing engine featuring Monte Carlo based position estimation, selective tree search based on Upper Confidence Bounds, and move valuation based on Elo ratings.
5. **523.xalancbmk_r** : XSLT processor for transforming XML documents into HTML, text, or other XML document types
6. **531.deepsjeng_r** : It is based on Deep Sjeng WC2008, the 2008 World Computer Speed-Chess Champion. Deep Sjeng is a rewrite of the older Sjeng-Free program, focused on obtaining the highest possible playing strength. It attempts to find the best move via a combination of alpha-beta tree searching, advanced move ordering, positional evaluation and heuristic forward pruning.

7. **557.xz_r** : Data compression. it incorporates pxz ; performs no file I/O other than reading the input; does all compression and decompression entirely in memory; and prefers generic portable routines rather than platform-specific routines. As usual for SPEC CPU®, the intent is to measure the compute-intensive portion of a real application, while minimizing IO; thereby focusing on the performance of the CPU, memory, and compiler.

F

Full DPS48E1 Slice functionality

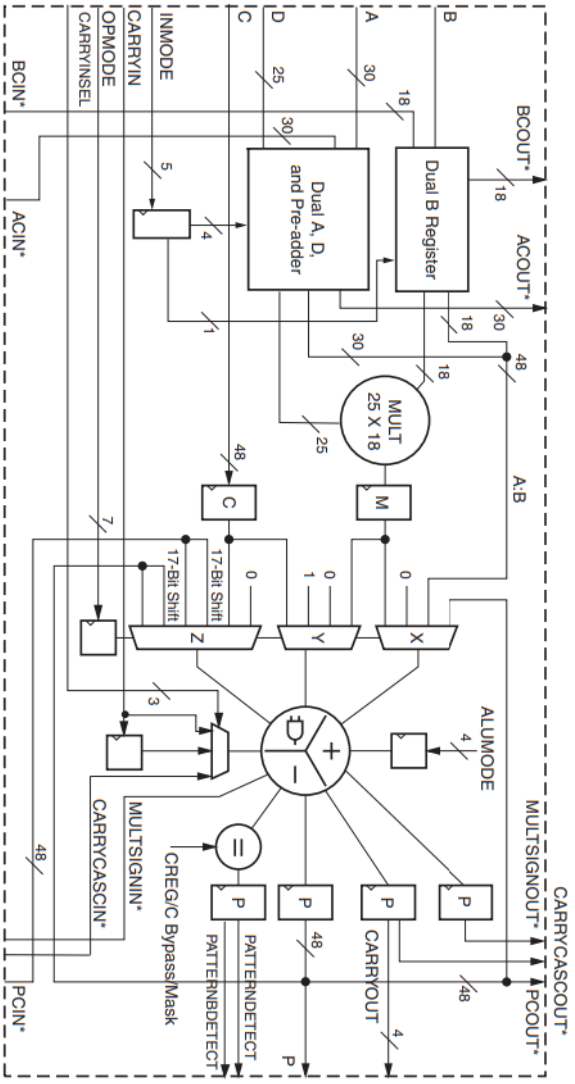


Figure F.1: Xilinx 7 Series DSP48E1 Slice [68]

G

ILP Modeling

The following sections elaborate on the ILP equations generated for estimating latency of SPFMA, DPFMA and FP Divsion units. The equations are categorized in three categories, namely,

- ‘Equal to’ : generated by putting in the type of operations in equation 4.1.
- ’Greater than’ : generated by plugging in the time dependency between operations in equation 4.2.
- ‘Less than’ : generated by plugging in the the total number of availbale resources of each type in equation 4.3.

The first line represents the objective function to be minimized. At last, the variables to be solved for are listed. In all the three cases (SPFAM, DPFMA, FP DIV) these are binary variables.

G.1. Single Precision Fused Multiply and Accumulate

```
min: +1 x5_1 +2 x5_2 +3 x5_3 +4 x5_4 +5 x5_5 -1; //Objective Function i.e
minimize the latency
//Category 1 Equations i.e ‘Equal to’ a.k.a unique operation
+ x1_1 + x1_2 + x1_3 + x1_4 + x1_5 = 1;
+ x2_1 + x2_2 + x2_3 + x2_4 + x2_5 = 1;
+ x3_1 + x3_2 + x3_3 + x3_4 + x3_5 = 1;
+ x4_1 + x4_2 + x4_3 + x4_4 + x4_5 = 1;
+ x5_1 + x5_2 + x5_3 + x5_4 + x5_5 = 1;
//Category 2 Equations i.e ’Greater than a.k.a time dependency
+ x1_1 +2 x1_2 +3 x1_3 +4 x1_4 +5 x1_5 >= 1;
+ x2_1 +2 x2_2 +3 x2_3 +4 x2_4 +5 x2_5 >= 1;
+ x3_1 +2 x3_2 +3 x3_3 +4 x3_4 +5 x3_5 >= 1;
+ x4_1 - x1_1 +2 x4_2 -2 x1_2 +3 x4_3 -3 x1_3
```

```
+4 x4_4 -4 x1_4 +5 x4_5 -5 x1_5 >= 1;
+ x4_1 - x2_1 +2 x4_2 -2 x2_2 +3 x4_3
-3 x2_3 +4 x4_4 -4 x2_4 +5 x4_5 -5 x2_5 >= 1;
+ x4_1 - x3_1 +2 x4_2 -2 x3_2 +3 x4_3 -3 x3_3
+4 x4_4 -4 x3_4 +5 x4_5 -5 x3_5 >= 1;
+ x5_1 - x4_1 +2 x5_2 -2 x4_2 +3 x5_3 -3 x4_3
+4 x5_4 -4 x4_4 +5 x5_5 -5 x4_5 >= 1;
//Category 2 Equations i.e 'less than a.k.a resource constraints
+ x2_1 + x3_1 <= 2;
+ x2_2 + x3_2 <= 2;
+ x2_3 + x3_3 <= 2;
+ x2_4 + x3_4 <= 2;
+ x1_1 + x4_1 + x5_1 <= 2;
+ x1_2 + x4_2 + x5_2 <= 2;
+ x1_3 + x4_3 + x5_3 <= 2;
+ x1_4 + x4_4 + x5_4 <= 2;
//Define the variables to be solved: binary
binary x1_1, x1_2, x1_3, x1_4, x1_5, x2_1,
x2_2, x2_3, x2_4, x2_5, x3_1, x3_2, x3_3,
x3_4, x3_5, x4_1, x4_2, x4_3, x4_4, x4_5,
x5_1, x5_2, x5_3, x5_4, x5_5;
```

G.2. Double Precision Fused Multiply and Accumulate

```
min: +1 x17_1 +2 x17_2 +3 x17_3 +4 x17_4 +5 x17_5 +6 x17_6 +7 x17_7
+8 x17_8 +9 x17_9 +10 x17_10 +11 x17_11 +12 x17_12 +13 x17_13
+14 x17_14 +15 x17_15 +16 x17_16 +17 x17_17 +18 x17_18
+19 x17_19 +20 x17_20 +21 x17_21 -1;
+ x1_1 + x1_2 + x1_3 + x1_4 + x1_5 + x1_6 + x1_7 + x1_8
+ x1_9 + x1_10 + x1_11 + x1_12 + x1_13 + x1_14 + x1_15
+ x1_16 + x1_17 + x1_18 + x1_19 + x1_20 + x1_21 = 1;
+ x2_1 + x2_2 + x2_3 + x2_4 + x2_5 + x2_6 + x2_7 + x2_8
+ x2_9 + x2_10 + x2_11 + x2_12 + x2_13 + x2_14 + x2_15
+ x2_16 + x2_17 + x2_18 + x2_19 + x2_20 + x2_21 = 1;
+ x3_1 + x3_2 + x3_3 + x3_4 + x3_5 + x3_6 + x3_7 + x3_8
+ x3_9 + x3_10 + x3_11 + x3_12 + x3_13 + x3_14 + x3_15
+ x3_16 + x3_17 + x3_18 + x3_19 + x3_20 + x3_21 = 1;
+ x4_1 + x4_2 + x4_3 + x4_4 + x4_5 + x4_6 + x4_7 + x4_8
+ x4_9 + x4_10 + x4_11 + x4_12 + x4_13 + x4_14 + x4_15
+ x4_16 + x4_17 + x4_18 + x4_19 + x4_20 + x4_21 = 1;
+ x5_1 + x5_2 + x5_3 + x5_4 + x5_5 + x5_6 + x5_7 + x5_8
+ x5_9 + x5_10 + x5_11 + x5_12 + x5_13 + x5_14 + x5_15
+ x5_16 + x5_17 + x5_18 + x5_19 + x5_20 + x5_21 = 1;
+ x6_1 + x6_2 + x6_3 + x6_4 + x6_5 + x6_6 + x6_7 + x6_8
```

```
+ x6_9 + x6_10 + x6_11 + x6_12 + x6_13 + x6_14 + x6_15
+ x6_16 + x6_17 + x6_18 + x6_19 + x6_20 + x6_21 = 1;
+ x7_1 + x7_2 + x7_3 + x7_4 + x7_5 + x7_6 + x7_7 + x7_8
+ x7_9 + x7_10 + x7_11 + x7_12 + x7_13 + x7_14 + x7_15
+ x7_16 + x7_17 + x7_18 + x7_19 + x7_20 + x7_21 = 1;
+ x8_1 + x8_2 + x8_3 + x8_4 + x8_5 + x8_6 + x8_7 + x8_8
+ x8_9 + x8_10 + x8_11 + x8_12 + x8_13 + x8_14 + x8_15
+ x8_16 + x8_17 + x8_18 + x8_19 + x8_20 + x8_21 = 1;
+ x9_1 + x9_2 + x9_3 + x9_4 + x9_5 + x9_6 + x9_7 + x9_8
+ x9_9 + x9_10 + x9_11 + x9_12 + x9_13 + x9_14 + x9_15
+ x9_16 + x9_17 + x9_18 + x9_19 + x9_20 + x9_21 = 1;
+ x10_1 + x10_2 + x10_3 + x10_4 + x10_5 + x10_6 + x10_7
+ x10_8 + x10_9 + x10_10 + x10_11 + x10_12 + x10_13
+ x10_14 + x10_15 + x10_16 + x10_17 + x10_18 + x10_19
+ x10_20 + x10_21 = 1;
+ x11_1 + x11_2 + x11_3 + x11_4 + x11_5 + x11_6 + x11_7
+ x11_8 + x11_9 + x11_10 + x11_11 + x11_12 + x11_13 + x11_14
+ x11_15 + x11_16 + x11_17 + x11_18 + x11_19 + x11_20 + x11_21 = 1;
+ x12_1 + x12_2 + x12_3 + x12_4 + x12_5 + x12_6 + x12_7
+ x12_8 + x12_9 + x12_10 + x12_11 + x12_12 + x12_13 + x12_14
+ x12_15 + x12_16 + x12_17 + x12_18 + x12_19 + x12_20 + x12_21 = 1;
+ x13_1 + x13_2 + x13_3 + x13_4 + x13_5 + x13_6 + x13_7 + x13_8
+ x13_9 + x13_10 + x13_11 + x13_12 + x13_13 + x13_14 + x13_15
+ x13_16 + x13_17 + x13_18 + x13_19 + x13_20 + x13_21 = 1;
+ x14_1 + x14_2 + x14_3 + x14_4 + x14_5 + x14_6 + x14_7 + x14_8
+ x14_9 + x14_10 + x14_11 + x14_12 + x14_13 + x14_14 + x14_15
+ x14_16 + x14_17 + x14_18 + x14_19 + x14_20 + x14_21 = 1;
+ x15_1 + x15_2 + x15_3 + x15_4 + x15_5 + x15_6 + x15_7 + x15_8
+ x15_9 + x15_10 + x15_11 + x15_12 + x15_13 + x15_14 + x15_15
+ x15_16 + x15_17 + x15_18 + x15_19 + x15_20 + x15_21 = 1;
+ x16_1 + x16_2 + x16_3 + x16_4 + x16_5 + x16_6 + x16_7 + x16_8
+ x16_9 + x16_10 + x16_11 + x16_12 + x16_13 + x16_14 + x16_15
+ x16_16 + x16_17 + x16_18 + x16_19 + x16_20 + x16_21 = 1;
+ x17_1 + x17_2 + x17_3 + x17_4 + x17_5 + x17_6 + x17_7 + x17_8
+ x17_9 + x17_10 + x17_11 + x17_12 + x17_13 + x17_14 + x17_15
+ x17_16 + x17_17 + x17_18 + x17_19 + x17_20 + x17_21 = 1;
+ x1_1 +2 x1_2 +3 x1_3 +4 x1_4 +5 x1_5 +6 x1_6 +7 x1_7 +8 x1_8
+9 x1_9 +10 x1_10 +11 x1_11 +12 x1_12 +13 x1_13 +14 x1_14 +15 x1_15
+16 x1_16 +17 x1_17 +18 x1_18 +19 x1_19 +20 x1_20 +21 x1_21 >= 1;
+ x2_1 +2 x2_2 +3 x2_3 +4 x2_4 +5 x2_5 +6 x2_6 +7 x2_7 +8 x2_8
+9 x2_9 +10 x2_10 +11 x2_11 +12 x2_12 +13 x2_13 +14 x2_14
+15 x2_15 +16 x2_16 +17 x2_17 +18 x2_18 +19 x2_19 +20 x2_20
+21 x2_21 >= 1;
+ x3_1 +2 x3_2 +3 x3_3 +4 x3_4 +5 x3_5 +6 x3_6 +7 x3_7 +8 x3_8
+9 x3_9 +10 x3_10 +11 x3_11 +12 x3_12 +13 x3_13 +14 x3_14
```

```
+15 x3_15 +16 x3_16 +17 x3_17 +18 x3_18 +19 x3_19 +20 x3_20 +21 x3_21 >= 1;
+ x4_1 +2 x4_2 +3 x4_3 +4 x4_4 +5 x4_5 +6 x4_6 +7 x4_7 +8 x4_8
+9 x4_9 +10 x4_10 +11 x4_11 +12 x4_12 +13 x4_13 +14 x4_14 +15 x4_15
+16 x4_16 +17 x4_17 +18 x4_18 +19 x4_19 +20 x4_20 +21 x4_21 >= 1;
+ x5_1 +2 x5_2 +3 x5_3 +4 x5_4 +5 x5_5 +6 x5_6 +7 x5_7 +8 x5_8
+9 x5_9 +10 x5_10 +11 x5_11 +12 x5_12 +13 x5_13 +14 x5_14
+15 x5_15 +16 x5_16 +17 x5_17 +18 x5_18 +19 x5_19 +20 x5_20
+21 x5_21 >= 1;
+ x6_1 +2 x6_2 +3 x6_3 +4 x6_4 +5 x6_5 +6 x6_6 +7 x6_7 +8 x6_8
+9 x6_9 +10 x6_10 +11 x6_11 +12 x6_12 +13 x6_13 +14 x6_14 +15 x6_15
+16 x6_16 +17 x6_17 +18 x6_18 +19 x6_19 +20 x6_20 +21 x6_21 >= 1;
+ x7_1 +2 x7_2 +3 x7_3 +4 x7_4 +5 x7_5 +6 x7_6 +7 x7_7 +8 x7_8
+9 x7_9 +10 x7_10 +11 x7_11 +12 x7_12 +13 x7_13 +14 x7_14 +15 x7_15
+16 x7_16 +17 x7_17 +18 x7_18 +19 x7_19 +20 x7_20 +21 x7_21 >= 1;
+ x8_1 +2 x8_2 +3 x8_3 +4 x8_4 +5 x8_5 +6 x8_6 +7 x8_7 +8 x8_8 +9 x8_9
+10 x8_10 +11 x8_11 +12 x8_12 +13 x8_13 +14 x8_14 +15 x8_15 +16 x8_16
+17 x8_17 +18 x8_18 +19 x8_19 +20 x8_20 +21 x8_21 >= 1;
+ x9_1 +2 x9_2 +3 x9_3 +4 x9_4 +5 x9_5 +6 x9_6 +7 x9_7 +8 x9_8
+9 x9_9 +10 x9_10 +11 x9_11 +12 x9_12 +13 x9_13 +14 x9_14 +15 x9_15
+16 x9_16 +17 x9_17 +18 x9_18 +19 x9_19 +20 x9_20 +21 x9_21 >= 1;
+ x10_1 +2 x10_2 +3 x10_3 +4 x10_4 +5 x10_5 +6 x10_6 +7 x10_7 +8 x10_8
+9 x10_9 +10 x10_10 +11 x10_11 +12 x10_12 +13 x10_13 +14 x10_14
+15 x10_15 +16 x10_16 +17 x10_17 +18 x10_18 +19 x10_19
+20 x10_20 +21 x10_21 >= 1;
+ x11_1 - x1_1 +2 x11_2 -2 x1_2 +3 x11_3 -3 x1_3 +4 x11_4 -4 x1_4 +5 x11_5
-5 x1_5 +6 x11_6 -6 x1_6 +7 x11_7 -7 x1_7 +8 x11_8 -8 x1_8 +9 x11_9 -9 x1_9
+10 x11_10 -10 x1_10 +11 x11_11 -11 x1_11 +12 x11_12 -12 x1_12 +13 x11_13
-13 x1_13 +14 x11_14 -14 x1_14 +15 x11_15 -15 x1_15 +16 x11_16 -16 x1_16
+17 x11_17 -17 x1_17 +18 x11_18 -18 x1_18 +19 x11_19 -19 x1_19 +20 x11_20
-20 x1_20 +21 x11_21 -21 x1_21 >= 1;
+ x11_1 - x2_1 +2 x11_2 -2 x2_2 +3 x11_3 -3 x2_3 +4 x11_4 -4 x2_4 +5 x11_5
-5 x2_5 +6 x11_6 -6 x2_6 +7 x11_7 -7 x2_7 +8 x11_8 -8 x2_8 +9 x11_9 -9 x2_9
+10 x11_10 -10 x2_10 +11 x11_11 -11 x2_11 +12 x11_12 -12 x2_12 +13 x11_13
-13 x2_13 +14 x11_14 -14 x2_14 +15 x11_15 -15 x2_15 +16 x11_16 -16 x2_16
+17 x11_17 -17 x2_17 +18 x11_18 -18 x2_18 +19 x11_19 -19 x2_19 +20 x11_20
-20 x2_20 +21 x11_21 -21 x2_21 >= 1;
+ x11_1 - x3_1 +2 x11_2 -2 x3_2 +3 x11_3 -3 x3_3 +4 x11_4 -4 x3_4 +5 x11_5
-5 x3_5 +6 x11_6 -6 x3_6 +7 x11_7 -7 x3_7 +8 x11_8 -8 x3_8 +9 x11_9 -9 x3_9
+10 x11_10 -10 x3_10 +11 x11_11 -11 x3_11 +12 x11_12 -12 x3_12 +13 x11_13
-13 x3_13 +14 x11_14 -14 x3_14 +15 x11_15 -15 x3_15 +16 x11_16 -16 x3_16
+17 x11_17 -17 x3_17 +18 x11_18 -18 x3_18 +19 x11_19 -19 x3_19 +20 x11_20
-20 x3_20 +21 x11_21 -21 x3_21 >= 1;
+ x12_1 - x4_1 +2 x12_2 -2 x4_2 +3 x12_3 -3 x4_3 +4 x12_4 -4 x4_4 +5 x12_5
-5 x4_5 +6 x12_6 -6 x4_6 +7 x12_7 -7 x4_7 +8 x12_8 -8 x4_8 +9 x12_9 -9 x4_9
+10 x12_10 -10 x4_10 +11 x12_11 -11 x4_11 +12 x12_12 -12 x4_12 +13 x12_13
```

```
-13 x4_13 +14 x12_14 -14 x4_14 +15 x12_15 -15 x4_15 +16 x12_16 -16 x4_16
+17 x12_17 -17 x4_17 +18 x12_18 -18 x4_18 +19 x12_19 -19 x4_19 +20 x12_20
-20 x4_20 +21 x12_21 -21 x4_21 >= 1;
+ x12_1 - x5_1 +2 x12_2 -2 x5_2 +3 x12_3 -3 x5_3 +4 x12_4 -4 x5_4 +5 x12_5
-5 x5_5 +6 x12_6 -6 x5_6 +7 x12_7 -7 x5_7 +8 x12_8 -8 x5_8 +9 x12_9 -9 x5_9
+10 x12_10 -10 x5_10 +11 x12_11 -11 x5_11 +12 x12_12 -12 x5_12 +13 x12_13
-13 x5_13 +14 x12_14 -14 x5_14 +15 x12_15 -15 x5_15 +16 x12_16 -16 x5_16
+17 x12_17 -17 x5_17 +18 x12_18 -18 x5_18 +19 x12_19 -19 x5_19 +20 x12_20
-20 x5_20 +21 x12_21 -21 x5_21 >= 1;
+ x12_1 - x6_1 +2 x12_2 -2 x6_2 +3 x12_3 -3 x6_3 +4 x12_4 -4 x6_4 +5 x12_5
-5 x6_5 +6 x12_6 -6 x6_6 +7 x12_7 -7 x6_7 +8 x12_8 -8 x6_8 +9 x12_9 -9 x6_9
+10 x12_10 -10 x6_10 +11 x12_11 -11 x6_11 +12 x12_12 -12 x6_12 +13 x12_13
-13 x6_13 +14 x12_14 -14 x6_14 +15 x12_15 -15 x6_15 +16 x12_16 -16 x6_16
+17 x12_17 -17 x6_17 +18 x12_18 -18 x6_18 +19 x12_19 -19 x6_19 +20 x12_20
-20 x6_20 +21 x12_21 -21 x6_21 >= 1;
+ x13_1 - x7_1 +2 x13_2 -2 x7_2 +3 x13_3 -3 x7_3 +4 x13_4 -4 x7_4 +5 x13_5
-5 x7_5 +6 x13_6 -6 x7_6 +7 x13_7 -7 x7_7 +8 x13_8 -8 x7_8 +9 x13_9 -9 x7_9
+10 x13_10 -10 x7_10 +11 x13_11 -11 x7_11 +12 x13_12 -12 x7_12 +13 x13_13
-13 x7_13 +14 x13_14 -14 x7_14 +15 x13_15 -15 x7_15 +16 x13_16 -16 x7_16
+17 x13_17 -17 x7_17 +18 x13_18 -18 x7_18 +19 x13_19 -19 x7_19 +20 x13_20
-20 x7_20 +21 x13_21 -21 x7_21 >= 1;
+ x13_1 - x8_1 +2 x13_2 -2 x8_2 +3 x13_3 -3 x8_3 +4 x13_4 -4 x8_4 +5 x13_5
-5 x8_5 +6 x13_6 -6 x8_6 +7 x13_7 -7 x8_7 +8 x13_8 -8 x8_8 +9 x13_9 -9 x8_9
+10 x13_10 -10 x8_10 +11 x13_11 -11 x8_11 +12 x13_12 -12 x8_12 +13 x13_13
-13 x8_13 +14 x13_14 -14 x8_14 +15 x13_15 -15 x8_15 +16 x13_16 -16 x8_16
+17 x13_17 -17 x8_17 +18 x13_18 -18 x8_18 +19 x13_19 -19 x8_19 +20 x13_20
-20 x8_20 +21 x13_21 -21 x8_21 >= 1;
+ x13_1 - x9_1 +2 x13_2 -2 x9_2 +3 x13_3 -3 x9_3 +4 x13_4 -4 x9_4 +5 x13_5
-5 x9_5 +6 x13_6 -6 x9_6 +7 x13_7 -7 x9_7 +8 x13_8 -8 x9_8 +9 x13_9 -9 x9_9
+10 x13_10 -10 x9_10 +11 x13_11 -11 x9_11 +12 x13_12 -12 x9_12 +13 x13_13
-13 x9_13 +14 x13_14 -14 x9_14 +15 x13_15 -15 x9_15 +16 x13_16 -16 x9_16
+17 x13_17 -17 x9_17 +18 x13_18 -18 x9_18 +19 x13_19 -19 x9_19 +20 x13_20
-20 x9_20 +21 x13_21 -21 x9_21 >= 1;
+ x14_1 - x10_1 +2 x14_2 -2 x10_2 +3 x14_3 -3 x10_3 +4 x14_4 -4 x10_4
+5 x14_5 -5 x10_5 +6 x14_6 -6 x10_6 +7 x14_7 -7 x10_7 +8 x14_8 -8 x10_8
+9 x14_9 -9 x10_9 +10 x14_10 -10 x10_10 +11 x14_11 -11 x10_11 +12 x14_12
-12 x10_12 +13 x14_13 -13 x10_13 +14 x14_14 -14 x10_14 +15 x14_15 -15 x10_15
+16 x14_16 -16 x10_16 +17 x14_17 -17 x10_17 +18 x14_18 -18 x10_18 +19 x14_19
-19 x10_19 +20 x14_20 -20 x10_20 +21 x14_21 -21 x10_21 >= 1;
+ x14_1 - x11_1 +2 x14_2 -2 x11_2 +3 x14_3 -3 x11_3 +4 x14_4 -4 x11_4
+5 x14_5 -5 x11_5 +6 x14_6 -6 x11_6 +7 x14_7 -7 x11_7 +8 x14_8 -8 x11_8
+9 x14_9 -9 x11_9 +10 x14_10 -10 x11_10 +11 x14_11 -11 x11_11 +12 x14_12
-12 x11_12 +13 x14_13 -13 x11_13 +14 x14_14 -14 x11_14 +15 x14_15
-15 x11_15 +16 x14_16 -16 x11_16 +17 x14_17 -17 x11_17 +18 x14_18
-18 x11_18 +19 x14_19 -19 x11_19 +20 x14_20
```

```
-20 x11_20 +21 x14_21 -21 x11_21 >= 1;
+ x14_1 - x12_1 +2 x14_2 -2 x12_2 +3 x14_3 -3 x12_3 +4 x14_4 -4 x12_4
+5 x14_5 -5 x12_5 +6 x14_6 -6 x12_6 +7 x14_7 -7 x12_7 +8 x14_8 -8 x12_8
+9 x14_9 -9 x12_9 +10 x14_10 -10 x12_10 +11 x14_11 -11 x12_11 +12 x14_12
-12 x12_12 +13 x14_13-13 x12_13 +14 x14_14 -14 x12_14 +15 x14_15 -15 x12_15
+16 x14_16 -16 x12_16 +17 x14_17 -17 x12_17 +18 x14_18 -18 x12_18 +19 x14_19
-19 x12_19 +20 x14_20 -20 x12_20 +21 x14_21 -21 x12_21 >= 1;
+ x15_1 - x13_1 +2 x15_2 -2 x13_2 +3 x15_3 -3 x13_3 +4 x15_4 -4 x13_4
+5 x15_5 -5 x13_5 +6 x15_6 -6 x13_6 +7 x15_7 -7 x13_7 +8 x15_8 -8 x13_8
+9 x15_9 -9 x13_9 +10 x15_10 -10 x13_10 +11 x15_11 -11 x13_11
+12 x15_12 -12 x13_12 +13 x15_13 -13 x13_13 +14 x15_14 -14 x13_14
+15 x15_15 -15 x13_15 +16 x15_16 -16 x13_16
+17 x15_17 -17 x13_17 +18 x15_18 -18 x13_18 +19 x15_19 -19 x13_19 +20 x15_20
-20 x13_20 +21 x15_21 -21 x13_21 >= 1;
+ x16_1 - x14_1 +2 x16_2 -2 x14_2 +3 x16_3 -3 x14_3 +4 x16_4 -4 x14_4
+5 x16_5 -5 x14_5 +6 x16_6 -6 x14_6 +7 x16_7 -7 x14_7 +8 x16_8 -8 x14_8
+9 x16_9 -9 x14_9 +10 x16_10 -10 x14_10 +11 x16_11 -11 x14_11 +12 x16_12
-12 x14_12 +13 x16_13 -13 x14_13 +14 x16_14 -14 x14_14 +15 x16_15 -15 x14_15
+16 x16_16 -16 x14_16 +17 x16_17 -17 x14_17 +18 x16_18 -18 x14_18 +19 x16_19
-19 x14_19 +20 x16_20 -20 x14_20 +21 x16_21 -21 x14_21 >= 1;
+ x17_1 - x15_1 +2 x17_2 -2 x15_2 +3 x17_3 -3 x15_3 +4 x17_4 -4 x15_4
+5 x17_5 -5 x15_5 +6 x17_6 -6 x15_6 +7 x17_7 -7 x15_7 +8 x17_8 -8 x15_8
+9 x17_9 -9 x15_9 +10 x17_10 -10 x15_10 +11 x17_11 -11 x15_11 +12 x17_12
-12 x15_12 +13 x17_13 -13 x15_13 +14 x17_14 -14 x15_14 +15 x17_15 -15 x15_15
+16 x17_16 -16 x15_16 +17 x17_17 -17 x15_17 +18 x17_18 -18 x15_18 +19 x17_19
-19 x15_19 +20 x17_20 -20 x15_20 +21 x17_21 -21 x15_21 >= 1;
+ x2_1 + x3_1 + x4_1 + x5_1 + x6_1 + x7_1 + x8_1 + x9_1 + x10_1 <= 9;
+ x2_2 + x3_2 + x4_2 + x5_2 + x6_2 + x7_2 + x8_2 + x9_2 + x10_2 <= 9;
+ x2_3 + x3_3 + x4_3 + x5_3 + x6_3 + x7_3 + x8_3 + x9_3 + x10_3 <= 9;
+ x2_4 + x3_4 + x4_4 + x5_4 + x6_4 + x7_4 + x8_4 + x9_4 + x10_4 <= 9;
+ x2_5 + x3_5 + x4_5 + x5_5 + x6_5 + x7_5 + x8_5 + x9_5 + x10_5 <= 9;
+ x2_6 + x3_6 + x4_6 + x5_6 + x6_6 + x7_6 + x8_6 + x9_6 + x10_6 <= 9;
+ x2_7 + x3_7 + x4_7 + x5_7 + x6_7 + x7_7 + x8_7 + x9_7 + x10_7 <= 9;
+ x2_8 + x3_8 + x4_8 + x5_8 + x6_8 + x7_8 + x8_8 + x9_8 + x10_8 <= 9;
+ x2_9 + x3_9 + x4_9 + x5_9 + x6_9 + x7_9 + x8_9 + x9_9 + x10_9 <= 9;
+ x2_10 + x3_10 + x4_10 + x5_10 + x6_10 + x7_10 + x8_10 + x9_10
+ x10_10 <= 9;
+ x2_11 + x3_11 + x4_11 + x5_11 + x6_11 + x7_11 + x8_11 + x9_11
+ x10_11 <= 9;
+ x2_12 + x3_12 + x4_12 + x5_12 + x6_12 + x7_12 + x8_12 + x9_12
+ x10_12 <= 9;
+ x2_13 + x3_13 + x4_13 + x5_13 + x6_13 + x7_13 + x8_13 + x9_13
+ x10_13 <= 9;
+ x2_14 + x3_14 + x4_14 + x5_14 + x6_14 + x7_14 + x8_14 + x9_14
+ x10_14 <= 9;
```

```
+ x2_15 + x3_15 + x4_15 + x5_15 + x6_15 + x7_15 + x8_15 + x9_15
+ x10_15 <= 9;
+ x2_16 + x3_16 + x4_16 + x5_16 + x6_16 + x7_16 + x8_16 + x9_16
+ x10_16 <= 9;
+ x2_17 + x3_17 + x4_17 + x5_17 + x6_17 + x7_17 + x8_17 + x9_17
+ x10_17 <= 9;
+ x2_18 + x3_18 + x4_18 + x5_18 + x6_18 + x7_18 + x8_18 + x9_18
+ x10_18 <= 9;
+ x2_19 + x3_19 + x4_19 + x5_19 + x6_19 + x7_19 + x8_19 + x9_19
+ x10_19 <= 9;
+ x2_20 + x3_20 + x4_20 + x5_20 + x6_20 + x7_20 + x8_20 + x9_20
+ x10_20 <= 9;
+ x1_1 + x11_1 + x12_1 + x13_1 + x14_1 + x15_1 + x16_1 + x17_1 <= 9;
+ x1_2 + x11_2 + x12_2 + x13_2 + x14_2 + x15_2 + x16_2 + x17_2 <= 9;
+ x1_3 + x11_3 + x12_3 + x13_3 + x14_3 + x15_3 + x16_3 + x17_3 <= 9;
+ x1_4 + x11_4 + x12_4 + x13_4 + x14_4 + x15_4 + x16_4 + x17_4 <= 9;
+ x1_5 + x11_5 + x12_5 + x13_5 + x14_5 + x15_5 + x16_5 + x17_5 <= 9;
+ x1_6 + x11_6 + x12_6 + x13_6 + x14_6 + x15_6 + x16_6 + x17_6 <= 9;
+ x1_7 + x11_7 + x12_7 + x13_7 + x14_7 + x15_7 + x16_7 + x17_7 <= 9;
+ x1_8 + x11_8 + x12_8 + x13_8 + x14_8 + x15_8 + x16_8 + x17_8 <= 9;
+ x1_9 + x11_9 + x12_9 + x13_9 + x14_9 + x15_9 + x16_9 + x17_9 <= 9;
+ x1_10 + x11_10 + x12_10 + x13_10 + x14_10 + x15_10 + x16_10 + x17_10 <= 9;
+ x1_11 + x11_11 + x12_11 + x13_11 + x14_11 + x15_11 + x16_11 + x17_11 <= 9;
+ x1_12 + x11_12 + x12_12 + x13_12 + x14_12 + x15_12 + x16_12 + x17_12 <= 9;
+ x1_13 + x11_13 + x12_13 + x13_13 + x14_13 + x15_13 + x16_13 + x17_13 <= 9;
+ x1_14 + x11_14 + x12_14 + x13_14 + x14_14 + x15_14 + x16_14 + x17_14 <= 9;
+ x1_15 + x11_15 + x12_15 + x13_15 + x14_15 + x15_15 + x16_15 + x17_15 <= 9;
+ x1_16 + x11_16 + x12_16 + x13_16 + x14_16 + x15_16 + x16_16 + x17_16 <= 9;
+ x1_17 + x11_17 + x12_17 + x13_17 + x14_17 + x15_17 + x16_17 + x17_17 <= 9;
+ x1_18 + x11_18 + x12_18 + x13_18 + x14_18 + x15_18 + x16_18 + x17_18 <= 9;
+ x1_19 + x11_19 + x12_19 + x13_19 + x14_19 + x15_19 + x16_19 + x17_19 <= 9;
+ x1_20 + x11_20 + x12_20 + x13_20 + x14_20 + x15_20 + x16_20 + x17_20 <= 9;
binary x1_1, x1_2, x1_3, x1_4, x1_5, x1_6, x1_7, x1_8, x1_9, x1_10,
x1_11, x1_12,x1_13, x1_14, x1_15, x1_16, x1_17, x1_18, x1_19, x1_20,
x1_21, x2_1, x2_2, x2_3, x2_4,x2_5, x2_6, x2_7, x2_8, x2_9, x2_10,
x2_11, x2_12, x2_13, x2_14, x2_15, x2_16, x2_17,x2_18, x2_19, x2_20,
x2_21, x3_1, x3_2, x3_3, x3_4, x3_5, x3_6, x3_7, x3_8, x3_9,
x3_10, x3_11, x3_12, x3_13, x3_14, x3_15, x3_16, x3_17, x3_18, x3_19,
x3_20, x3_21, x4_1, x4_2, x4_3, x4_4, x4_5, x4_6, x4_7, x4_8, x4_9, x4_10,
x4_11, x4_12, x4_13, x4_14, x4_15, x4_16, x4_17, x4_18, x4_19, x4_20, x4_21,
x5_1, x5_2, x5_3, x5_4, x5_5, x5_6, x5_7, x5_8, x5_9, x5_10, x5_11, x5_12,
x5_13, x5_14, x5_15, x5_16, x5_17, x5_18, x5_19, x5_20, x5_21, x6_1, x6_2,
x6_3, x6_4, x6_5, x6_6, x6_7, x6_8, x6_9, x6_10, x6_11, x6_12, x6_13, x6_14,
x6_15, x6_16, x6_17, x6_18, x6_19, x6_20, x6_21, x7_1, x7_2, x7_3, x7_4,
x7_5, x7_6, x7_7, x7_8, x7_9, x7_10, x7_11, x7_12, x7_13, x7_14, x7_15,
```

```
x7_16, x7_17, x7_18, x7_19, x7_20, x7_21, x8_1, x8_2, x8_3, x8_4, x8_5,
x8_6, x8_7, x8_8, x8_9, x8_10, x8_11, x8_12, x8_13, x8_14, x8_15, x8_16,
x8_17, x8_18, x8_19, x8_20, x8_21, x9_1, x9_2, x9_3, x9_4, x9_5, x9_6,
x9_7, x9_8, x9_9, x9_10, x9_11, x9_12, x9_13, x9_14, x9_15, x9_16, x9_17,
x9_18, x9_19, x9_20, x9_21, x10_1, x10_2, x10_3,x10_4, x10_5, x10_6,
x10_7, x10_8, x10_9, x10_10, x10_11, x10_12, x10_13, x10_14, x10_15,
x10_16, x10_17, x10_18, x10_19, x10_20, x10_21, x11_1, x11_2, x11_3,
x11_4, x11_5, x11_6, x11_7, x11_8, x11_9, x11_10, x11_11, x11_12,
x11_13, x11_14, x11_15, x11_16, x11_17, x11_18, x11_19, x11_20,
x11_21, x12_1, x12_2, x12_3, x12_4, x12_5, x12_6, x12_7, x12_8, x12_9,
x12_10, x12_11, x12_12, x12_13, x12_14, x12_15, x12_16, x12_17,
x12_18, x12_19, x12_20, x12_21, x13_1, x13_2, x13_3, x13_4, x13_5,
x13_6, x13_7, x13_8, x13_9, x13_10, x13_11, x13_12, x13_13, x13_14,
x13_15, x13_16, x13_17, x13_18, x13_19, x13_20, x13_21, x14_1, x14_2,
x14_3, x14_4, x14_5, x14_6, x14_7, x14_8, x14_9, x14_10, x14_11,
x14_12, x14_13, x14_14, x14_15, x14_16, x14_17, x14_18, x14_19,
x14_20, x14_21, x15_1, x15_2, x15_3, x15_4, x15_5, x15_6, x15_7,
x15_8, x15_9, x15_10, x15_11, x15_12, x15_13, x15_14, x15_15, x15_16,
x15_17, x15_18, x15_19, x15_20, x15_21, x16_1, x16_2, x16_3, x16_4,
x16_5, x16_6, x16_7, x16_8, x16_9, x16_10, x16_11, x16_12, x16_13,
x16_14, x16_15, x16_16, x16_17, x16_18, x16_19, x16_20, x16_21, x17_1,
x17_2, x17_3, x17_4, x17_5, x17_6, x17_7, x17_8, x17_9, x17_10, x17_11,
x17_12, x17_13, x17_14, x17_15, x17_16, x17_17, x17_18, x17_19, x17_20,
x17_21;
```

G.3. Floating Point Division: Division by Convergence

```
min: +1 x14_1 +2 x14_2 +3 x14_3 +4 x14_4 +5 x14_5 +6 x14_6 +7 x14_7
+8 x14_8 +9 x14_9 +10 x14_10 +11 x14_11 +12 x14_12 +13 x14_13
+14 x14_14 +15 x14_15 +16 x14_16 +17 x14_17 +18 x14_18 +19 x14_19
+20 x14_20 +21 x14_21 +22 x14_22 +23 x14_23 +24 x14_24 +25 x14_25
+26 x14_26 +27 x14_27 +28 x14_28 +29 x14_29 +30 x14_30 +31 x14_31 -1;
+ x1_1 + x1_2 + x1_3 + x1_4 + x1_5 + x1_6 + x1_7 + x1_8 + x1_9 + x1_10
+ x1_11 + x1_12 + x1_13 + x1_14 + x1_15 + x1_16 + x1_17 + x1_18 + x1_19
+ x1_20 + x1_21 + x1_22 + x1_23 + x1_24 + x1_25 + x1_26 + x1_27 + x1_28
+ x1_29 + x1_30 + x1_31 = 1;
+ x2_1 + x2_2 + x2_3 + x2_4 + x2_5 + x2_6 + x2_7 + x2_8 + x2_9 + x2_10
+ x2_11 + x2_12 + x2_13 + x2_14 + x2_15 + x2_16 + x2_17 + x2_18 + x2_19
+ x2_20 + x2_21 + x2_22 + x2_23 + x2_24 + x2_25 + x2_26 + x2_27 + x2_28
+ x2_29 + x2_30 + x2_31 = 1;
+ x3_1 + x3_2 + x3_3 + x3_4 + x3_5 + x3_6 + x3_7 + x3_8 + x3_9 + x3_10
+ x3_11 + x3_12 + x3_13 + x3_14 + x3_15 + x3_16 + x3_17 + x3_18 + x3_19
+ x3_20 + x3_21 + x3_22 + x3_23 + x3_24 + x3_25 + x3_26 + x3_27 + x3_28
+ x3_29 + x3_30 + x3_31 = 1;
```

```
+ x4_1 + x4_2 + x4_3 + x4_4 + x4_5 + x4_6 + x4_7 + x4_8 + x4_9 + x4_10
+ x4_11 + x4_12 + x4_13 + x4_14 + x4_15 + x4_16 + x4_17 + x4_18 + x4_19
+ x4_20 + x4_21 + x4_22 + x4_23 + x4_24 + x4_25 + x4_26 + x4_27 + x4_28
+ x4_29 + x4_30 + x4_31 = 1;
+ x5_1 + x5_2 + x5_3 + x5_4 + x5_5 + x5_6 + x5_7 + x5_8 + x5_9 + x5_10
+ x5_11 + x5_12 + x5_13 + x5_14 + x5_15 + x5_16 + x5_17 + x5_18 + x5_19
+ x5_20 + x5_21 + x5_22 + x5_23 + x5_24 + x5_25 + x5_26 + x5_27 + x5_28
+ x5_29 + x5_30 + x5_31 = 1;
+ x6_1 + x6_2 + x6_3 + x6_4 + x6_5 + x6_6 + x6_7 + x6_8 + x6_9 + x6_10
+ x6_11 + x6_12 + x6_13 + x6_14 + x6_15 + x6_16 + x6_17 + x6_18 + x6_19
+ x6_20 + x6_21 + x6_22 + x6_23 + x6_24 + x6_25 + x6_26 + x6_27 + x6_28
+ x6_29 + x6_30 + x6_31 = 1;
+ x7_1 + x7_2 + x7_3 + x7_4 + x7_5 + x7_6 + x7_7 + x7_8 + x7_9 + x7_10
+ x7_11 + x7_12 + x7_13 + x7_14 + x7_15 + x7_16 + x7_17 + x7_18 + x7_19
+ x7_20 + x7_21 + x7_22 + x7_23 + x7_24 + x7_25 + x7_26 + x7_27 + x7_28
+ x7_29 + x7_30 + x7_31 = 1;
+ x8_1 + x8_2 + x8_3 + x8_4 + x8_5 + x8_6 + x8_7 + x8_8 + x8_9 + x8_10
+ x8_11 + x8_12 + x8_13 + x8_14 + x8_15 + x8_16 + x8_17 + x8_18 + x8_19
+ x8_20 + x8_21 + x8_22 + x8_23 + x8_24 + x8_25 + x8_26 + x8_27 + x8_28
+ x8_29 + x8_30 + x8_31 = 1;
+ x9_1 + x9_2 + x9_3 + x9_4 + x9_5 + x9_6 + x9_7 + x9_8 + x9_9 + x9_10
+ x9_11 + x9_12 + x9_13 + x9_14 + x9_15 + x9_16 + x9_17 + x9_18 + x9_19
+ x9_20 + x9_21 + x9_22 + x9_23 + x9_24 + x9_25 + x9_26 + x9_27 + x9_28
+ x9_29 + x9_30 + x9_31 = 1;
+ x10_1 + x10_2 + x10_3 + x10_4 + x10_5 + x10_6 + x10_7 + x10_8 + x10_9
+ x10_10 + x10_11 + x10_12 + x10_13 + x10_14 + x10_15 + x10_16 + x10_17
+ x10_18 + x10_19 + x10_20 + x10_21 + x10_22 + x10_23 + x10_24 + x10_25
\+ x10_26 + x10_27 + x10_28 + x10_29 + x10_30 + x10_31 = 1;
+ x11_1 + x11_2 + x11_3 + x11_4 + x11_5 + x11_6 + x11_7 + x11_8 + x11_9
+ x11_10 + x11_11 + x11_12 + x11_13 + x11_14 + x11_15 + x11_16 + x11_17
+ x11_18 + x11_19 + x11_20 + x11_21 + x11_22 + x11_23 + x11_24 + x11_25
+ x11_26 + x11_27 + x11_28 + x11_29 + x11_30 + x11_31 = 1;
+ x12_1 + x12_2 + x12_3 + x12_4 + x12_5 + x12_6 + x12_7 + x12_8 + x12_9
+ x12_10 + x12_11 + x12_12 + x12_13 + x12_14 + x12_15 + x12_16 + x12_17
+ x12_18 + x12_19 + x12_20 + x12_21 + x12_22 + x12_23 + x12_24 + x12_25
+ x12_26 + x12_27 + x12_28 + x12_29 + x12_30 + x12_31 = 1;
+ x13_1 + x13_2 + x13_3 + x13_4 + x13_5 + x13_6 + x13_7 + x13_8 + x13_9
+ x13_10 + x13_11 + x13_12 + x13_13 + x13_14 + x13_15 + x13_16 + x13_17
+ x13_18 + x13_19 + x13_20 + x13_21 + x13_22 + x13_23 + x13_24 + x13_25
+ x13_26 + x13_27 + x13_28 + x13_29 + x13_30 + x13_31 = 1;
+ x14_1 + x14_2 + x14_3 + x14_4 + x14_5 + x14_6 + x14_7 + x14_8 + x14_9
+ x14_10 + x14_11 + x14_12 + x14_13 + x14_14 + x14_15 + x14_16 + x14_17
+ x14_18 + x14_19 + x14_20 + x14_21 + x14_22 + x14_23 + x14_24 + x14_25
+ x14_26 + x14_27 + x14_28 + x14_29 + x14_30 + x14_31 = 1;
+ x1_1 +2 x1_2 +3 x1_3 +4 x1_4 +5 x1_5 +6 x1_6 +7 x1_7 +8 x1_8 +9 x1_9
```

```
+10 x1_10 +11 x1_11 +12 x1_12 +13 x1_13 +14 x1_14 +15 x1_15 +16 x1_16
+17 x1_17 +18 x1_18 +19 x1_19 +20 x1_20 +21 x1_21 +22 x1_22 +23 x1_23
+24 x1_24 +25 x1_25 +26 x1_26 +27 x1_27 +28 x1_28 +29 x1_29 +30 x1_30
+31 x1_31 >= 1;
+ x2_1 +2 x2_2 +3 x2_3 +4 x2_4 +5 x2_5 +6 x2_6 +7 x2_7 +8 x2_8 +9 x2_9
+10 x2_10 +11 x2_11 +12 x2_12 +13 x2_13 +14 x2_14 +15 x2_15 +16 x2_16
+17 x2_17 +18 x2_18 +19 x2_19 +20 x2_20 +21 x2_21 +22 x2_22 +23 x2_23
+24 x2_24 +25 x2_25 +26 x2_26 +27 x2_27 +28 x2_28 +29 x2_29 +30 x2_30
+31 x2_31 >= 1;
+ x3_1 - x1_1 +2 x3_2 -2 x1_2 +3 x3_3 -3 x1_3 +4 x3_4 -4 x1_4 +5 x3_5
-5 x1_5 +6 x3_6 -6 x1_6 +7 x3_7 -7 x1_7 +8 x3_8 -8 x1_8 +9 x3_9 -9 x1_9
+10 x3_10 -10 x1_10 +11 x3_11 -11 x1_11 +12 x3_12 -12 x1_12 +13 x3_13
-13 x1_13 +14 x3_14 -14 x1_14 +15 x3_15 -15 x1_15 +16 x3_16 -16 x1_16
+17 x3_17 -17 x1_17 +18 x3_18 -18 x1_18 +19 x3_19 -19 x1_19 +20 x3_20
-20 x1_20 +21 x3_21 -21 x1_21 +22 x3_22 -22 x1_22 +23 x3_23 -23 x1_23
+24 x3_24 -24 x1_24 +25 x3_25 -25 x1_25 +26 x3_26 -26 x1_26 +27 x3_27
-27 x1_27 +28 x3_28 -28 x1_28 +29 x3_29 -29 x1_29 +30 x3_30 -30 x1_30
+31 x3_31 -31 x1_31 >= 1;
+ x4_1 - x3_1 +2 x4_2 -2 x3_2 +3 x4_3 -3 x3_3 +4 x4_4 -4 x3_4 +5 x4_5
-5 x3_5 +6 x4_6 -6 x3_6 +7 x4_7 -7 x3_7 +8 x4_8 -8 x3_8 +9 x4_9 -9 x3_9
+10 x4_10 -10 x3_10 +11 x4_11 -11 x3_11 +12 x4_12 -12 x3_12 +13 x4_13
-13 x3_13 +14 x4_14 -14 x3_14 +15 x4_15 -15 x3_15 +16 x4_16 -16 x3_16
+17 x4_17 -17 x3_17 +18 x4_18 -18 x3_18 +19 x4_19 -19 x3_19 +20 x4_20
-20 x3_20 +21 x4_21 -21 x3_21 +22 x4_22 -22 x3_22 +23 x4_23 -23 x3_23
+24 x4_24 -24 x3_24 +25 x4_25 -25 x3_25 +26 x4_26 -26 x3_26 +27 x4_27
-27 x3_27 +28 x4_28 -28 x3_28 +29 x4_29 -29 x3_29 +30 x4_30 -30 x3_30
+31 x4_31 -31 x3_31 >= 1;
+ x5_1 - x2_1 +2 x5_2 -2 x2_2 +3 x5_3 -3 x2_3 +4 x5_4 -4 x2_4 +5 x5_5
-5 x2_5 +6 x5_6 -6 x2_6 +7 x5_7 -7 x2_7 +8 x5_8 -8 x2_8 +9 x5_9 -9 x2_9
+10 x5_10 -10 x2_10 +11 x5_11 -11 x2_11 +12 x5_12 -12 x2_12 +13 x5_13
-13 x2_13 +14 x5_14 -14 x2_14 +15 x5_15 -15 x2_15 +16 x5_16 -16 x2_16
+17 x5_17 -17 x2_17 +18 x5_18 -18 x2_18 +19 x5_19 -19 x2_19 +20 x5_20
-20 x2_20 +21 x5_21 -21 x2_21 +22 x5_22 -22 x2_22 +23 x5_23 -23 x2_23
+24 x5_24 -24 x2_24 +25 x5_25 -25 x2_25 +26 x5_26 -26 x2_26 +27 x5_27
-27 x2_27 +28 x5_28 -28 x2_28 +29 x5_29 -29 x2_29 +30 x5_30 -30 x2_30
+31 x5_31 -31 x2_31 >= 1;
+ x5_1 - x3_1 +2 x5_2 -2 x3_2 +3 x5_3 -3 x3_3 +4 x5_4 -4 x3_4 +5 x5_5
-5 x3_5 +6 x5_6 -6 x3_6 +7 x5_7 -7 x3_7 +8 x5_8 -8 x3_8 +9 x5_9 -9 x3_9
+10 x5_10 -10 x3_10 +11 x5_11 -11 x3_11 +12 x5_12 -12 x3_12 +13 x5_13
-13 x3_13 +14 x5_14 -14 x3_14 +15 x5_15 -15 x3_15 +16 x5_16 -16 x3_16
+17 x5_17 -17 x3_17 +18 x5_18 -18 x3_18 +19 x5_19 -19 x3_19 +20 x5_20
-20 x3_20 +21 x5_21 -21 x3_21 +22 x5_22 -22 x3_22 +23 x5_23 -23 x3_23
+24 x5_24 -24 x3_24 +25 x5_25 -25 x3_25 +26 x5_26 -26 x3_26 +27 x5_27
-27 x3_27 +28 x5_28 -28 x3_28 +29 x5_29 -29 x3_29 +30 x5_30 -30 x3_30
+31 x5_31 -31 x3_31 >= 1;
```

```
+ x6_1 - x4_1 +2 x6_2 -2 x4_2 +3 x6_3 -3 x4_3 +4 x6_4 -4 x4_4 +5 x6_5
-5 x4_5 +6 x6_6 -6 x4_6 +7 x6_7 -7 x4_7 +8 x6_8 -8 x4_8 +9 x6_9 -9 x4_9
+10 x6_10 -10 x4_10 +11 x6_11 -11 x4_11 +12 x6_12 -12 x4_12 +13 x6_13
-13 x4_13 +14 x6_14 -14 x4_14 +15 x6_15 -15 x4_15 +16 x6_16 -16 x4_16
+17 x6_17 -17 x4_17 +18 x6_18 -18 x4_18 +19 x6_19 -19 x4_19 +20 x6_20
-20 x4_20 +21 x6_21 -21 x4_21 +22 x6_22 -22 x4_22 +23 x6_23 -23 x4_23
+24 x6_24 -24 x4_24 +25 x6_25 -25 x4_25 +26 x6_26 -26 x4_26 +27 x6_27
-27 x4_27 +28 x6_28 -28 x4_28 +29 x6_29 -29 x4_29 +30 x6_30 -30 x4_30
+31 x6_31 -31 x4_31 >= 1;
+ x7_1 - x6_1 +2 x7_2 -2 x6_2 +3 x7_3 -3 x6_3 +4 x7_4 -4 x6_4 +5 x7_5
-5 x6_5 +6 x7_6 -6 x6_6 +7 x7_7 -7 x6_7 +8 x7_8 -8 x6_8 +9 x7_9 -9 x6_9
+10 x7_10 -10 x6_10 +11 x7_11 -11 x6_11 +12 x7_12 -12 x6_12 +13 x7_13
-13 x6_13 +14 x7_14 -14 x6_14 +15 x7_15 -15 x6_15 +16 x7_16 -16 x6_16
+17 x7_17 -17 x6_17 +18 x7_18 -18 x6_18 +19 x7_19 -19 x6_19 +20 x7_20
-20 x6_20 +21 x7_21 -21 x6_21 +22 x7_22 -22 x6_22 +23 x7_23 -23 x6_23
+24 x7_24 -24 x6_24 +25 x7_25 -25 x6_25 +26 x7_26 -26 x6_26 +27 x7_27
-27 x6_27 +28 x7_28 -28 x6_28 +29 x7_29 -29 x6_29 +30 x7_30 -30 x6_30
+31 x7_31 -31 x6_31 >= 1;
+ x8_1 - x5_1 +2 x8_2 -2 x5_2 +3 x8_3 -3 x5_3 +4 x8_4 -4 x5_4 +5 x8_5
 -5 x5_5 +6 x8_6 -6 x5_6 +7 x8_7 -7 x5_7 +8 x8_8 -8 x5_8 +9 x8_9 -9 x5_9
 +10 x8_10 -10 x5_10 +11 x8_11 -11 x5_11 +12 x8_12 -12 x5_12 +13 x8_13
 -13 x5_13 +14 x8_14 -14 x5_14 +15 x8_15 -15 x5_15 +16 x8_16 -16 x5_16
 +17 x8_17 -17 x5_17 +18 x8_18 -18 x5_18 +19 x8_19 -19 x5_19 +20 x8_20
 -20 x5_20 +21 x8_21 -21 x5_21 +22 x8_22 -22 x5_22 +23 x8_23 -23 x5_23
 +24 x8_24 -24 x5_24 +25 x8_25 -25 x5_25 +26 x8_26 -26 x5_26 +27 x8_27
 -27 x5_27 +28 x8_28 -28 x5_28 +29 x8_29 -29 x5_29 +30 x8_30 -30 x5_30
 +31 x8_31 -31 x5_31 >= 1;
+ x8_1 - x6_1 +2 x8_2 -2 x6_2 +3 x8_3 -3 x6_3 +4 x8_4 -4 x6_4 +5 x8_5
-5 x6_5 +6 x8_6 -6 x6_6 +7 x8_7 -7 x6_7 +8 x8_8 -8 x6_8 +9 x8_9 -9 x6_9
+10 x8_10 -10 x6_10 +11 x8_11 -11 x6_11 +12 x8_12 -12 x6_12 +13 x8_13
-13 x6_13 +14 x8_14 -14 x6_14 +15 x8_15 -15 x6_15 +16 x8_16 -16 x6_16
+17 x8_17 -17 x6_17 +18 x8_18 -18 x6_18 +19 x8_19 -19 x6_19 +20 x8_20
-20 x6_20 +21 x8_21 -21 x6_21 +22 x8_22 -22 x6_22 +23 x8_23 -23 x6_23
+24 x8_24 -24 x6_24 +25 x8_25 -25 x6_25 +26 x8_26 -26 x6_26 +27 x8_27
-27 x6_27 +28 x8_28 -28 x6_28 +29 x8_29 -29 x6_29 +30 x8_30 -30 x6_30
+31 x8_31 -31 x6_31 >= 1;
+ x9_1 - x7_1 +2 x9_2 -2 x7_2 +3 x9_3 -3 x7_3 +4 x9_4 -4 x7_4 +5 x9_5
-5 x7_5 +6 x9_6 -6 x7_6 +7 x9_7 -7 x7_7 +8 x9_8 -8 x7_8 +9 x9_9 -9 x7_9
+10 x9_10 -10 x7_10 +11 x9_11 -11 x7_11 +12 x9_12 -12 x7_12 +13 x9_13
-13 x7_13 +14 x9_14 -14 x7_14 +15 x9_15 -15 x7_15 +16 x9_16 -16 x7_16
+17 x9_17 -17 x7_17 +18 x9_18 -18 x7_18 +19 x9_19 -19 x7_19 +20 x9_20
-20 x7_20 +21 x9_21 -21 x7_21 +22 x9_22 -22 x7_22 +23 x9_23 -23 x7_23
+24 x9_24 -24 x7_24 +25 x9_25 -25 x7_25 +26 x9_26 -26 x7_26 +27 x9_27
-27 x7_27 +28 x9_28 -28 x7_28 +29 x9_29 -29 x7_29 +30 x9_30 -30 x7_30
+31 x9_31 -31 x7_31 >= 1;
```

```
+ x10_1 - x9_1 +2 x10_2 -2 x9_2 +3 x10_3 -3 x9_3 +4 x10_4 -4 x9_4
+5 x10_5 -5 x9_5 +6 x10_6 -6 x9_6 +7 x10_7 -7 x9_7 +8 x10_8 -8 x9_8
+9 x10_9 -9 x9_9 +10 x10_10 -10 x9_10 +11 x10_11 -11 x9_11 +12 x10_12
-12 x9_12 +13 x10_13 -13 x9_13 +14 x10_14 -14 x9_14 +15 x10_15 -15 x9_15
+16 x10_16 -16 x9_16 +17 x10_17 -17 x9_17 +18 x10_18 -18 x9_18 +19 x10_19
-19 x9_19 +20 x10_20 -20 x9_20 +21 x10_21 -21 x9_21 +22 x10_22 -22 x9_22
+23 x10_23 -23 x9_23 +24 x10_24 -24 x9_24 +25 x10_25 -25 x9_25 +26 x10_26
-26 x9_26 +27 x10_27 -27 x9_27 +28 x10_28 -28 x9_28 +29 x10_29 -29 x9_29
+30 x10_30 -30 x9_30 +31 x10_31 -31 x9_31 >= 1;
+ x11_1 - x8_1 +2 x11_2 -2 x8_2 +3 x11_3 -3 x8_3 +4 x11_4 -4 x8_4
+5 x11_5 -5 x8_5 +6 x11_6 -6 x8_6 +7 x11_7 -7 x8_7 +8 x11_8 -8 x8_8
+9 x11_9 -9 x8_9 +10 x11_10 -10 x8_10 +11 x11_11 -11 x8_11 +12 x11_12
-12 x8_12 +13 x11_13 -13 x8_13 +14 x11_14 -14 x8_14 +15 x11_15 -15 x8_15
+16 x11_16 -16 x8_16 +17 x11_17 -17 x8_17 +18 x11_18 -18 x8_18 +19 x11_19
-19 x8_19 +20 x11_20 -20 x8_20 +21 x11_21 -21 x8_21 +22 x11_22 -22 x8_22
+23 x11_23 -23 x8_23 +24 x11_24 -24 x8_24 +25 x11_25 -25 x8_25 +26 x11_26
-26 x8_26 +27 x11_27 -27 x8_27 +28 x11_28 -28 x8_28 +29 x11_29 -29 x8_29
+30 x11_30 -30 x8_30 +31 x11_31 -31 x8_31 >= 1;
+ x11_1 - x9_1 +2 x11_2 -2 x9_2 +3 x11_3 -3 x9_3 +4 x11_4 -4 x9_4 +5 x11_5
-5 x9_5 +6 x11_6 -6 x9_6 +7 x11_7 -7 x9_7 +8 x11_8 -8 x9_8 +9 x11_9 -9 x9_9
+10 x11_10 -10 x9_10 +11 x11_11 -11 x9_11 +12 x11_12 -12 x9_12 +13 x11_13
-13 x9_13 +14 x11_14 -14 x9_14 +15 x11_15 -15 x9_15 +16 x11_16 -16 x9_16
+17 x11_17 -17 x9_17 +18 x11_18 -18 x9_18 +19 x11_19 -19 x9_19 +20 x11_20
-20 x9_20 +21 x11_21 -21 x9_21 +22 x11_22 -22 x9_22 +23 x11_23 -23 x9_23
+24 x11_24 -24 x9_24 +25 x11_25 -25 x9_25 +26 x11_26 -26 x9_26 +27 x11_27
-27 x9_27 +28 x11_28 -28 x9_28 +29 x11_29 -29 x9_29 +30 x11_30 -30 x9_30
+31 x11_31 -31 x9_31 >= 1;
+ x12_1 - x10_1 +2 x12_2 -2 x10_2 +3 x12_3 -3 x10_3 +4 x12_4 -4 x10_4
+5 x12_5 -5 x10_5 +6 x12_6 -6 x10_6 +7 x12_7 -7 x10_7 +8 x12_8 -8 x10_8
+9 x12_9 -9 x10_9 +10 x12_10 -10 x10_10 +11 x12_11 -11 x10_11 +12 x12_12
-12 x10_12 +13 x12_13 -13 x10_13 +14 x12_14 -14 x10_14 +15 x12_15 -15 x10_15
+16 x12_16 -16 x10_16 +17 x12_17 -17 x10_17 +18 x12_18 -18 x10_18 +19 x12_19
-19 x10_19 +20 x12_20 -20 x10_20 +21 x12_21 -21 x10_21 +22 x12_22 -22 x10_22
+23 x12_23 -23 x10_23 +24 x12_24 -24 x10_24 +25 x12_25 -25 x10_25 +26 x12_26
-26 x10_26 +27 x12_27 -27 x10_27 +28 x12_28 -28 x10_28 +29 x12_29 -29 x10_29
+30 x12_30 -30 x10_30 +31 x12_31 -31 x10_31 >= 1;
+ x13_1 - x11_1 +2 x13_2 -2 x11_2 +3 x13_3 -3 x11_3 +4 x13_4 -4 x11_4
+5 x13_5 -5 x11_5 +6 x13_6 -6 x11_6 +7 x13_7 -7 x11_7 +8 x13_8 -8 x11_8
+9 x13_9 -9 x11_9 +10 x13_10 -10 x11_10 +11 x13_11 -11 x11_11 +12 x13_12
-12 x11_12 +13 x13_13 -13 x11_13 +14 x13_14 -14 x11_14 +15 x13_15
-15 x11_15 +16 x13_16 -16 x11_16 +17 x13_17 -17 x11_17 +18 x13_18
-18 x11_18 +19 x13_19 -19 x11_19 +20 x13_20 -20 x11_20 +21 x13_21
-21 x11_21 +22 x13_22 -22 x11_22 +23 x13_23 -23 x11_23 +24 x13_24
-24 x11_24 +25 x13_25 -25 x11_25 +26 x13_26 -26 x11_26 +27 x13_27
-27 x11_27 +28 x13_28 -28 x11_28 +29 x13_29 -29 x11_29 +30 x13_30
```

```
-30 x11_30 +31 x13_31 -31 x11_31 >= 1;
+ x13_1 - x12_1 +2 x13_2 -2 x12_2 +3 x13_3 -3 x12_3 +4 x13_4 -4 x12_4
+5 x13_5 -5 x12_5 +6 x13_6 -6 x12_6 +7 x13_7 -7 x12_7 +8 x13_8 -8 x12_8
+9 x13_9 -9 x12_9 +10 x13_10 -10 x12_10 +11 x13_11 -11 x12_11 +12 x13_12
-12 x12_12 +13 x13_13 -13 x12_13 +14 x13_14 -14 x12_14 +15 x13_15
-15 x12_15 +16 x13_16 -16 x12_16 +17 x13_17 -17 x12_17 +18 x13_18
-18 x12_18 +19 x13_19 -19 x12_19 +20 x13_20 -20 x12_20 +21 x13_21
-21 x12_21 +22 x13_22 -22 x12_22 +23 x13_23 -23 x12_23 +24 x13_24
-24 x12_24 +25 x13_25 -25 x12_25 +26 x13_26 -26 x12_26 +27 x13_27
-27 x12_27 +28 x13_28 -28 x12_28 +29 x13_29 -29 x12_29 +30 x13_30
-30 x12_30 +31 x13_31 -31 x12_31 >= 1;
+ x14_1 - x13_1 +2 x14_2 -2 x13_2 +3 x14_3 -3 x13_3 +4 x14_4
-4 x13_4 +5 x14_5 -5 x13_5 +6 x14_6 -6 x13_6 +7 x14_7 -7 x13_7
+8 x14_8 -8 x13_8 +9 x14_9 -9 x13_9 +10 x14_10 -10 x13_10 +11 x14_11
-11 x13_11 +12 x14_12 -12 x13_12 +13 x14_13 -13 x13_13 +14 x14_14
-14 x13_14 +15 x14_15 -15 x13_15 +16 x14_16 -16 x13_16 +17 x14_17
-17 x13_17 +18 x14_18 -18 x13_18 +19 x14_19 -19 x13_19 +20 x14_20
-20 x13_20 +21 x14_21 -21 x13_21 +22 x14_22 -22 x13_22 +23 x14_23
-23 x13_23 +24 x14_24 -24 x13_24 +25 x14_25 -25 x13_25 +26 x14_26
-26 x13_26 +27 x14_27 -27 x13_27 +28 x14_28 -28 x13_28 +29 x14_29
-29 x13_29 +30 x14_30 -30 x13_30 +31 x14_31 -31 x13_31 >= 1;
+ x1_1 + x2_1 + x4_1 + x5_1 + x7_1 + x8_1 + x10_1 + x11_1 + x13_1 <= 4;
+ x1_2 + x2_2 + x4_2 + x5_2 + x7_2 + x8_2 + x10_2 + x11_2 + x13_2 <= 4;
+ x1_3 + x2_3 + x4_3 + x5_3 + x7_3 + x8_3 + x10_3 + x11_3 + x13_3 <= 4;
+ x1_4 + x2_4 + x4_4 + x5_4 + x7_4 + x8_4 + x10_4 + x11_4 + x13_4 <= 4;
+ x1_5 + x2_5 + x4_5 + x5_5 + x7_5 + x8_5 + x10_5 + x11_5 + x13_5 <= 4;
+ x1_6 + x2_6 + x4_6 + x5_6 + x7_6 + x8_6 + x10_6 + x11_6 + x13_6 <= 4;
+ x1_7 + x2_7 + x4_7 + x5_7 + x7_7 + x8_7 + x10_7 + x11_7 + x13_7 <= 4;
+ x1_8 + x2_8 + x4_8 + x5_8 + x7_8 + x8_8 + x10_8 + x11_8 + x13_8 <= 4;
+ x1_9 + x2_9 + x4_9 + x5_9 + x7_9 + x8_9 + x10_9 + x11_9 + x13_9 <= 4;
+ x1_10 + x2_10 + x4_10 + x5_10 + x7_10 + x8_10 + x10_10 + x11_10
+ x13_10 <= 4;
+ x1_11 + x2_11 + x4_11 + x5_11 + x7_11 + x8_11 + x10_11 + x11_11
+ x13_11 <= 4;
+ x1_12 + x2_12 + x4_12 + x5_12 + x7_12 + x8_12 + x10_12 + x11_12
+ x13_12 <= 4;
+ x1_13 + x2_13 + x4_13 + x5_13 + x7_13 + x8_13 + x10_13 + x11_13
+ x13_13 <= 4;
+ x1_14 + x2_14 + x4_14 + x5_14 + x7_14 + x8_14 + x10_14 + x11_14
+ x13_14 <= 4;
+ x1_15 + x2_15 + x4_15 + x5_15 + x7_15 + x8_15 + x10_15 + x11_15
+ x13_15 <= 4;
+ x1_16 + x2_16 + x4_16 + x5_16 + x7_16 + x8_16 + x10_16 + x11_16
+ x13_16 <= 4;
+ x1_17 + x2_17 + x4_17 + x5_17 + x7_17 + x8_17 + x10_17 + x11_17
```

```
+ x13_17 <= 4;
+ x1_18 + x2_18 + x4_18 + x5_18 + x7_18 + x8_18 + x10_18 + x11_18
+ x13_18 <= 4;
+ x1_19 + x2_19 + x4_19 + x5_19 + x7_19 + x8_19 + x10_19 + x11_19
+ x13_19 <= 4;
+ x1_20 + x2_20 + x4_20 + x5_20 + x7_20 + x8_20 + x10_20 + x11_20
+ x13_20 <= 4;
+ x1_21 + x2_21 + x4_21 + x5_21 + x7_21 + x8_21 + x10_21 + x11_21
+ x13_21 <= 4;
+ x1_22 + x2_22 + x4_22 + x5_22 + x7_22 + x8_22 + x10_22 + x11_22
+ x13_22 <= 4;
+ x1_23 + x2_23 + x4_23 + x5_23 + x7_23 + x8_23 + x10_23 + x11_23
+ x13_23 <= 4;
+ x1_24 + x2_24 + x4_24 + x5_24 + x7_24 + x8_24 + x10_24 + x11_24
+ x13_24 <= 4;
+ x1_25 + x2_25 + x4_25 + x5_25 + x7_25 + x8_25 + x10_25 + x11_25
+ x13_25 <= 4;
+ x1_26 + x2_26 + x4_26 + x5_26 + x7_26 + x8_26 + x10_26 + x11_26
+ x13_26 <= 4;
+ x1_27 + x2_27 + x4_27 + x5_27 + x7_27 + x8_27 + x10_27 + x11_27
+ x13_27 <= 4;
+ x1_28 + x2_28 + x4_28 + x5_28 + x7_28 + x8_28 + x10_28 + x11_28
+ x13_28 <= 4;
+ x1_29 + x2_29 + x4_29 + x5_29 + x7_29 + x8_29 + x10_29 + x11_29
+ x13_29 <= 4;
+ x1_30 + x2_30 + x4_30 + x5_30 + x7_30 + x8_30 + x10_30 + x11_30
+ x13_30 <= 4;
+ x3_1 + x6_1 + x9_1 + x12_1 + x14_1 <= 4;
+ x3_2 + x6_2 + x9_2 + x12_2 + x14_2 <= 4;
+ x3_3 + x6_3 + x9_3 + x12_3 + x14_3 <= 4;
+ x3_4 + x6_4 + x9_4 + x12_4 + x14_4 <= 4;
+ x3_5 + x6_5 + x9_5 + x12_5 + x14_5 <= 4;
+ x3_6 + x6_6 + x9_6 + x12_6 + x14_6 <= 4;
+ x3_7 + x6_7 + x9_7 + x12_7 + x14_7 <= 4;
+ x3_8 + x6_8 + x9_8 + x12_8 + x14_8 <= 4;
+ x3_9 + x6_9 + x9_9 + x12_9 + x14_9 <= 4;
+ x3_10 + x6_10 + x9_10 + x12_10 + x14_10 <= 4;
+ x3_11 + x6_11 + x9_11 + x12_11 + x14_11 <= 4;
+ x3_12 + x6_12 + x9_12 + x12_12 + x14_12 <= 4;
+ x3_13 + x6_13 + x9_13 + x12_13 + x14_13 <= 4;
+ x3_14 + x6_14 + x9_14 + x12_14 + x14_14 <= 4;
+ x3_15 + x6_15 + x9_15 + x12_15 + x14_15 <= 4;
+ x3_16 + x6_16 + x9_16 + x12_16 + x14_16 <= 4;
+ x3_17 + x6_17 + x9_17 + x12_17 + x14_17 <= 4;
+ x3_18 + x6_18 + x9_18 + x12_18 + x14_18 <= 4;
```

```
+ x3_19 + x6_19 + x9_19 + x12_19 + x14_19 <= 4;
+ x3_20 + x6_20 + x9_20 + x12_20 + x14_20 <= 4;
+ x3_21 + x6_21 + x9_21 + x12_21 + x14_21 <= 4;
+ x3_22 + x6_22 + x9_22 + x12_22 + x14_22 <= 4;
+ x3_23 + x6_23 + x9_23 + x12_23 + x14_23 <= 4;
+ x3_24 + x6_24 + x9_24 + x12_24 + x14_24 <= 4;
+ x3_25 + x6_25 + x9_25 + x12_25 + x14_25 <= 4;
+ x3_26 + x6_26 + x9_26 + x12_26 + x14_26 <= 4;
+ x3_27 + x6_27 + x9_27 + x12_27 + x14_27 <= 4;
+ x3_28 + x6_28 + x9_28 + x12_28 + x14_28 <= 4;
+ x3_29 + x6_29 + x9_29 + x12_29 + x14_29 <= 4;
+ x3_30 + x6_30 + x9_30 + x12_30 + x14_30 <= 4;
binary x1_1, x1_2, x1_3, x1_4, x1_5, x1_6, x1_7, x1_8, x1_9,
x1_10, x1_11, x1_12, x1_13, x1_14, x1_15, x1_16, x1_17, x1_18,
x1_19, x1_20, x1_21, x1_22, x1_23, x1_24, x1_25, x1_26, x1_27,
x1_28, x1_29, x1_30, x1_31, x2_1, x2_2, x2_3, x2_4, x2_5, x2_6,
x2_7, x2_8, x2_9, x2_10, x2_11, x2_12, x2_13, x2_14, x2_15,
x2_16, x2_17, x2_18, x2_19, x2_20, x2_21, x2_22, x2_23, x2_24,
x2_25, x2_26, x2_27, x2_28, x2_29, x2_30, x2_31, x3_1, x3_2,
x3_3, x3_4, x3_5, x3_6, x3_7, x3_8, x3_9, x3_10, x3_11, x3_12,
x3_13, x3_14, x3_15, x3_16, x3_17, x3_18, x3_19, x3_20, x3_21,
x3_22, x3_23, x3_24, x3_25, x3_26, x3_27, x3_28, x3_29, x3_30,
x3_31, x4_1, x4_2, x4_3, x4_4, x4_5, x4_6, x4_7, x4_8, x4_9,
x4_10, x4_11, x4_12, x4_13, x4_14, x4_15, x4_16, x4_17, x4_18,
x4_19, x4_20, x4_21, x4_22, x4_23, x4_24, x4_25, x4_26, x4_27,
x4_28, x4_29, x4_30, x4_31, x5_1, x5_2, x5_3, x5_4, x5_5, x5_6,
x5_7, x5_8, x5_9, x5_10, x5_11, x5_12, x5_13, x5_14, x5_15, x5_16,
x5_17, x5_18, x5_19, x5_20, x5_21, x5_22, x5_23, x5_24, x5_25,
x5_26, x5_27, x5_28, x5_29, x5_30, x5_31, x6_1, x6_2, x6_3, x6_4,
x6_5, x6_6, x6_7, x6_8, x6_9, x6_10, x6_11, x6_12, x6_13, x6_14,
x6_15, x6_16, x6_17, x6_18, x6_19, x6_20, x6_21, x6_22, x6_23,
x6_24, x6_25, x6_26, x6_27, x6_28, x6_29, x6_30, x6_31, x7_1,
x7_2, x7_3, x7_4, x7_5, x7_6, x7_7, x7_8, x7_9, x7_10, x7_11,
x7_12, x7_13, x7_14, x7_15, x7_16, x7_17, x7_18, x7_19, x7_20,
x7_21, x7_22, x7_23, x7_24, x7_25, x7_26, x7_27, x7_28, x7_29,
x7_30, x7_31, x8_1, x8_2, x8_3, x8_4, x8_5, x8_6, x8_7, x8_8,
x8_9, x8_10, x8_11, x8_12, x8_13, x8_14, x8_15, x8_16, x8_17,
x8_18, x8_19, x8_20, x8_21, x8_22, x8_23, x8_24, x8_25, x8_26,
x8_27, x8_28, x8_29, x8_30, x8_31, x9_1, x9_2, x9_3, x9_4, x9_5,
x9_6, x9_7, x9_8, x9_9, x9_10, x9_11, x9_12, x9_13, x9_14, x9_15,
x9_16, x9_17, x9_18, x9_19, x9_20, x9_21, x9_22, x9_23, x9_24,
x9_25, x9_26, x9_27, x9_28, x9_29, x9_30, x9_31, x10_1, x10_2,
x10_3, x10_4, x10_5, x10_6, x10_7, x10_8, x10_9, x10_10, x10_11,
x10_12, x10_13, x10_14, x10_15, x10_16, x10_17, x10_18, x10_19,
x10_20, x10_21, x10_22, x10_23, x10_24, x10_25, x10_26, x10_27,
```

```
x10_28, x10_29, x10_30, x10_31, x11_1, x11_2, x11_3, x11_4, x11_5,
x11_6, x11_7, x11_8, x11_9, x11_10, x11_11, x11_12, x11_13, x11_14,
x11_15, x11_16, x11_17, x11_18, x11_19, x11_20, x11_21, x11_22,
 x11_23, x11_24, x11_25, x11_26, x11_27, x11_28, x11_29, x11_30,
 x11_31, x12_1, x12_2, x12_3, x12_4, x12_5, x12_6, x12_7, x12_8,
 x12_9, x12_10, x12_11, x12_12, x12_13, x12_14, x12_15, x12_16,
 x12_17, x12_18, x12_19, x12_20, x12_21, x12_22, x12_23, x12_24,
 x12_25, x12_26, x12_27, x12_28, x12_29, x12_30, x12_31, x13_1,
 x13_2, x13_3, x13_4, x13_5, x13_6, x13_7, x13_8, x13_9, x13_10,
 x13_11, x13_12, x13_13, x13_14, x13_15, x13_16, x13_17, x13_18,
 x13_19, x13_20, x13_21, x13_22, x13_23, x13_24, x13_25, x13_26,
 x13_27, x13_28, x13_29, x13_30, x13_31, x14_1, x14_2, x14_3,
 x14_4, x14_5, x14_6, x14_7, x14_8, x14_9, x14_10, x14_11, x14_12,
 x14_13, x14_14, x14_15, x14_16, x14_17, x14_18, x14_19, x14_20,
 x14_21, x14_22, x14_23, x14_24, x14_25, x14_26, x14_27, x14_28,
 x14_29, x14_30, x14_31;
```

H

gXR5 Specifications

Component	Attribute	Baseline Simulator	Validated Simulator
MinorCPU model			
IntALU	Operation Latency (Cycles)	3	2
IntDiv	Operation Latency(Cycles)	33	19
ReadMem FU	Operation Latency	4	3
WriteMem FU	Operation Latency(Cycles)	2	1
Decode Unit	Input Width, Buffer Size	1	4
Fetch2 Unit	Input Buffer Size	2	4
Execute Unit	Input Width	1	4
Branch Predictor	Type	Tournament	Multiperspective Perceptron
Caches			
L1 Data	Data Latency	4	1
L1 Data	Response Latency	2	2
L1 Data	Clusivity	Mostly Exclusive	Mostly Inclusive
L1 Instruction	Clusivity	Mostly Exclusive	Mostly Inclusive

Table H.1: Selected attributes of Baseline and Validated Simulator (against Sifive Unleashed

Component	Attribute	Baseline Simulator	Validated Simulator
MinorCPU model			
IntDiv	Operation Latency	33	16
FloatMultAcc	Operation Latency	3	5
FloatDiv	Operation Latency	5	9
Branch Predictor	Type	Tournament	Gshare

Table H.2: Selected attributes of Baseline and Validated Simulator (against Rocket System)