EPFL

Thèse n° 8990

Building Chips Faster: Hardware-Compiler Co-Design for Accelerated RTL Simulation

Présentée le 6 septembre 2023

Faculté informatique et communications
Laboratoire d'informatique à très grande échelle
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Sahand KASHANI

Acceptée sur proposition du jury

Prof. E. Bugnion, président du jury
Prof. J. R. Larus, directeur de thèse
Prof. J. C. Hoe, rapporteur
Prof. S. Beamer, rapporteur
Prof. P. Ienne, rapporteur

École
polytechnique
fédérale
de Lausanne

2023

Acknowledgements

There are many people who deserve my heartfelt thanks. I only regret that I cannot mention them all by name in these few paragraphs, but I will try.

I want to first thank my advisor, *Prof. James Larus*, for accepting to take me on as a grad student and for his patience with me over the past six years. I am surprised I even managed to make it this far as there were numerous stretches of time where I was making no progress for months, to the point where I was convinced Jim would fire me. However, Jim kept surprising me by (1) not firing me, and (2) saying something unexpected that somehow changed the course of my PhD.

- The first was when, two years into my PhD, I recognized that the project I was working on was leading nowhere. To my surprise, Jim walked in my office one day and just said "you should go on an internship". I did not feel I deserved to go on an internship, but Jim had suggested it himself, and I was happy about the prospect of working on something different for a few months. It turned out that working on something unrelated to my PhD helped clear my mind: I had come up with an original idea of my own—totally different from what I was working on before and during my internship—which I wanted to instead pursue for my PhD, and which Jim miraculously agreed to[1].
- The second was when I recognized that the way I was attempting to bring my idea to life was not the right approach. I was very upset at this stage as it had been four years since I started the PhD, with still no results to show in comparison to other students. I expressed my concern to Jim who, to my surprise, then said something along the following lines:

  > In grad school people tend to see the outliers, those who seem to succeed without much difficulty. However, remember that those are outliers and that most people do not go through grad school easily. It takes many years for students to understand how to do research: 50% of the difficulty in a PhD is finding a good problem to solve, the other 50% is solving that problem.

What Jim said about 50% of the difficulty in a PhD being finding a good problem to solve stuck with me. I was pursuing my idea with the mindset: "what could I do if I had X?" However, I realized that this problem was too open-ended and that I should instead be thinking: "what is the specific problem I am trying to solve, and what do I need to solve it?" I still remember the afternoon of July 16th 2021 when I was brainstorming with my office mate to figure out how to pivot my project: we came up with a variation of my original idea which we found applicable to

[1]The original idea I had pitched to Jim was wild and I was surprised when he agreed to me even pursuing it!

the problem of accelerated RTL simulation. The “Manticore” project then became the main topic of my thesis, thankfully led to good results, and enabled me to graduate. It took a long time, but Jim’s patience with me allowed me to make the mistakes I had to make to figure out the real problem I had to solve. Few professors would give a student the freedom to work on a wild idea he has no prior experience in, and even fewer would tolerate not having results for such a long period of time. So thank you Jim for the trust you placed in me to finally figure things out, for your help putting Manticore’s best foot forward, and for your valuable advice over the years.

VLSC-DCSL

I am fortunate to have shared the corridor with the many nice (and entertaining) people who formed the VLSC-DCSL “consortium”. Thank you Adrien, Boris, Charly, David, Dmitrii, Endri, George, Jonas, Kostas, Mahyar, Marios, Mia, Neelu, Rui, Sam, Stanko, Stuart, and Yuchen for the fun times over the years, both in Switzerland and abroad. You all made the corridor a pleasant place to spend time.

I owe a great expression of gratitude to my dear friend and colleague *Mahyar Emami* for his invaluable help on the Manticore project. It is after brainstorming with him that I was able to finally understand how to pivot the project to what ultimately led to this thesis. Manticore was a massive project which, four years into my PhD, was unlikely I’d be able to pursue to completion by myself. Fortunately, Mahyar accepted to work with me on this project: blending each other’s working styles led us to produce an output which was more than double what either of us would have achieved on our own. We had no idea our collaboration would continue for the next 1.5 years before the first Manticore prototype would be completed, and that it would also lead Mahyar down the path to his next project. I would not have made it here without his help. He witnessed my day-to-day struggles, checking in on me when he sensed something was wrong. I will always remember his preferred approach to get me through these episodes: the acute sense of sarcasm he had developed over the years. I will miss his morning jokes about me not working enough (probably untrue), being on vacation all the time (certainly untrue), and constantly falling asleep at the office (definitely true).

David Aksun was the first student from VLSC who I had met. While originally a very private person, I discovered over time that we actually share many interests, in particular our passion for tiramisu and Pokémon music. I will remember the many evenings we spent working in the office with Pokémon music playing over the large speakers under his desk. Another distinct event I remember was on January 1st 2021, during the COVID-19 pandemic, when David called me at 8 p.m. and insisted we do a speedrun of Pokémon Sapphire; good times. His knack for sarcasm towards the end of his PhD greatly contributed to the fun times in the office alongside Mahyar.

I thank *Endri Bezati* for the countless long conversations we shared throughout the years. Though we did not technically work together, Endri never missed an opportunity to pass by my office on his way back from the kitchen during his 9 a.m. morning coffee ritual. Endri is a true geek who loves to explain his ideas and the hacks he does to trick FPGA tools into implementing them. His relationship with FPGAs and their tooling is truly one of love and hate, and I learnt something

each time we spoke. I also want to thank him for the helpful advice he gave me over the years, both on the professional and personal front.

Adrien Ghosn's peculiar sense of humor was a source of laughter multiple times each day, and I wish we had met earlier in life. The funny thing is that Adrien & I actually started as freshmen at EPFL at the same time, but met only during grad school. I thank him for trying to get me through my mental blockades by encouraging me to go to the CrossFit gym with him every morning at 6:15 a.m. Though the physical effort helped relax my mind so I could think better during the day, it also came at the expense of me being sleepy in the office for multiple years. Thankfully, Adrien was a generous soul who let me enjoy high-quality sleep on his office "pouf". I also thank him for our "Zooburger rituals", his humorous social experiments on people passing through our corridor, the numerous fun times in Seattle and Cambridge, and for inviting me to join him and his family for Christmas in 2022.

I thank *Prof. Edouard Bugnion* for breaking the ice and introducing me to Jim when I first joined grad school; I would not have joined VLSC were it not for his original introduction. I'd also like to thank Ed for his feedback, in particular for his insightful comments on the Manticore project that changed how we pitched this work and led to a stronger story and paper. I also always appreciated the summer BBQs at Ed's place and his anecdotes about management challenges in industry, academia, and even in politics given his multiple hats over the years.

Finally, I want to thank *Maggy Church* and *Tania Epars* for their help with all administrative tasks in the office over the years. They always knew who to call, were on top of everything, and ensured things go smoothly for us students. I feel we often do not thank them enough for their kind efforts behind the scences.

LAP

Though I spent my time in grad school as part of the VLSC-DCSL consortium, my first interaction with a lab at EPFL dates back to 2012 when I started working at LAP as a TA for Prof. Paolo Ienne's computer architecture course. LAP was a very welcoming place where I met great people with whom I felt at home, and so ended up staying during my Masters and frequently passed by during grad school. I want to thank all current and former members of LAP with whom I overlapped over the past 11 years: Ali, Ana, André, Andrea, Andrew, Aya, Canberk, Chantal, David, Grace, Joao, Jovan, Lana, Louis, Lucas, Mikhail, Mirjana, Mohammed, Nikola, Nithin, Paolo, René, Robert, Stefan, Théo, and Xavier.

I want to extend a special thanks to some of the founding members of the "LAP Beer Club", who accepted me within their inner circle[2], and with whom I shared multiple fun evenings at LAP, or over dinner in Lausanne or in Bern: Ana, Andrea, Grace, Lana, Mikhail, and Stefan.

I quickly became good friends with *Grace Zgheib* and *Ana Petkovska*, who I first met in 2012 while TAing Paolo's course. I am honored that they entrusted me with their office plants after they

[2]Even though I don't drink beer.

both graduated. I thank them for their support over the years, and the many memorable dinners at our favorite Chinese restaurant in Lausanne. In particular, I express my deepest gratitude to Grace—certainly the most honest and correct person I have ever met—for constantly looking out for me during grad school. Despite her busy schedule, she always found time to chat and talk some sense into me when things were not advancing so that I could pull myself together. I'm also very grateful for her hosting me whenever I attended the FPGA conference, the great road trip while I was on my internship, and the fun time at the amusement park in California.

Lana Josipović is the person at LAP with perhaps the most magnetic personality, which is difficult to describe in words. All I can say is that her constant smiling, energy, and hugs always cheered me up whenever we met. Thanks for always being up for a meetup in Zurich when I passed by, and for being my go-to person for discussing a specific topic over the years.

I shared many memorable times with *Andrea Guerrieri*, *Stefan Nikolic*, and *Mikhail Asiatici*. I am fortunate to have met Andrea during my Master thesis. We somehow "clicked" and had multiple hours-long chats in his office over the years. I was very proud to hear that he would soon continue on as a professor in Sion; he totally deserves it. Stefan is one of the most patient and considerate people I know and always invited me to the spontaneous events he organized. I will remember his excessively-high tolerance while drinking, and his attempts to find countless arguments to convince you of something[3]. Though we did not interact as much after the COVID-19 pandemic, Mikhail was always a great friend who came back once a year to join us on the "LAP Beer Club" events. I will remember how these three guys would drag a carton of beer from Lausanne to Bern, and back, when they realized we can't drink our own beer at a bar in Bern.

I want to thank *René Beuchat*, *André Guignard*, and *Chantal Schneeberger* for the countless coffee breaks, laughs, and crossword puzzles in the LAP kitchen. I owe René many thanks for passing on his knowledge of embedded systems to me, and for entrenching within me the idea of keeping an engineering notebook while I work. This notebook has helped me far more than he can imagine. I was pleasantly surprised by René's trust in my skills when he asked me to (1) help design the curriculum for the new "System-on-Chip Programming Project" course in 2016, and (2) contribute two chapters to a book he was writing in 2019 [12]. Writing this book turned out to be an interesting experience alongside my colleagues and friends René, Andrea, and Florian. I will remember the many evenings we spent writing during the COVID-19 pandemic, and the memorable two days spent at René's place to finalize everything.

I thank *Prof. Paolo Ienne* for hosting me at LAP during my undergrad, inviting me to LAP's internal events, and for encouraging me to apply to grad school.

DSLAB

I thank *Rishabh Iyer* and *Arseniy Zaostrovnykh* for their friendship and camaraderie over the years, especially with everything sports-related. Rishabh is easily the most competitive person I know: he always found a way to persuade me to try out some new fitness challenge he had set for himself

[3] Very entertaining to watch.

after a deadline, especially when it's really not a good time for me to be participating in one. Most of these challenges naturally involved Rishabh's next-door office neighbor, Arseniy, with whom Rishabh had a long-standing, not-so-secret rivalry with respect to fitness acomplishments. I will miss my many interactions and fun moments with them, in particular watching how they each deliver their witty puns to each other.

Beyond a great friend, Rishabh has also been a great mentor over the years, even though we both started the PhD at the same time. Being the most academically-inclined person I know, he has developed the skill of distilling a paper's contributions into a few simple, digestable sentences. His own papers were also exceptionally clear, and he always insisted on nailing the abstract and introduction (two sections which I honestly found sucked in most papers). I will always remember how Rishabh casually asked to read the introduction of a paper submission I was working on, asked if I would be open to "disruptive feedback", then literally sat next to me 30h before the deadline to pair-program a new paper from scratch which was significantly better. I am eternally grateful for everything he taught me that day, and ever since Rishabh & I have spoken freely. I am glad we got a point in our friendship where we can put politeness aside when the bottom line is helping the other person out.

PARSA

I spent a significant amount of time with *Ognjen "Ogi" Glamočanin* and *Siddharth "Sid" Gupta*, both at EPFL and abroad at conferences (Ogi at FPGA conferences, and Sid at systems ones).

Ogi and Sid were two core members of the "Indian truck" group—alongside Mahyar, Adrien, Rishabh, and Xinrui—on Thursday afternoons. Many laughs and anecdotes were exchanged over the years while we waited for the truck to open, and when we got back to the VLSC-DCSL or PARSA kitchen to eat together.

Sid somehow convinced me to go on a trip to frozen Sweden/Finland with him and Ahmet for New Year's eve 2023. I will remember the nights sleeping at the airport gates, their constant ranting about the cold[4], and the memorable meals and events we shared when we finally got to the destination. Despite my initial reluctance to go on this trip, I am glad they managed to convince me to join them. Though we knew each other well before, I am happy to say our friendship only got better afterwards.

EDIC

I am grateful to everyone from EDIC who started the PhD journey around the same time as me, and with whom I stayed close throughout: Atri, Bharath, Bogdan, Jakab, Karen, Mariam, Matteo, Merlin, Novak, Paritosh, Sandra, Sena, Thanasis, and Xinrui.

[4]They willingly chose to go on this trip.

Special thanks to Jakab & Matteo for inviting me to their dinner parties and climbing sessions; to Bernd, Bharath, Bogdan, Karen, Mariam, and Sandra for the many walks, movies, lunches, dinners, and scientific/political discussions; to Merlin for the fun times in Seattle during our internships, for inviting me to his wedding, and for his great advice while searching for a job; and to Sena for the laughs and for our numerous Super Mario Bros competitions[5].

JSC CrossFit

Outside those who I met at EPFL, my social life was complemented with my daily interactions with the "Early Birds" at JSC CrossFit. I will miss everyone there with whom I shared a 45m gym session almost every weekday morning at 6:15 a.m. since March 2019. Thank you Adrien, Axel, Carlo, Christophe, Cyril, Dani, David, Filipe, Guillaume, Krista, Marc, Marisol, Melody, Rafael, and Sara for being motivated enough to wake up early and make the Early Bird session the first scheduled event of the day. This would not have been possible without the support of JSC's amazing coaches: Cyrille, Fabien, Fabrice, Hugo, Jon, Liz, Liza, Luana, and Romain. If you are tired and can't wake up, believe me that joining a session with these guys at 6:15 a.m. will do the job. Their training and encouragements helped me get stronger over the years and I hope we can stay in touch in the next steps of my fitness journey.

Family

I'd like to thank my parents, *Amir* and *Fariba*, for giving me the education needed to write this thesis in the first place, and for their support throughout the past 31 years. In particular, I am very much indebted towards my father for all the sacrifices he made so I could make it here. While I can never pay them back for all they have done for me, I can only attempt to pay their efforts forward to help others in the next stage of my life.

I'd also like to thank my twin brother *Sepand*, with whom I shared pretty much all experiences in life until we came to EPFL, for his support throughout the years. While we now experience totally different things in life, it was always funny to come back home and rant about things, only to hear very much similar rants coming back. Though our experiences had technically diverged, it seems that—in practice—they had converged to similar complaints and conclusions. I guess some things never actually change in life.

Funding

Finally, I must thank the many individuals who financially supported my PhD journey, in particular the Swiss taxpayers whose taxes helped fund EPFL and my work.

Lausanne, August 4th, 2023 S.K.

[5]She beat me every time.

Abstract

The demise of Moore's Law and Dennard scaling has resulted in diminishing performance gains for general-purpose processors, and so has prompted a surge in academic and commercial interest for hardware accelerators. Specialized hardware has already redefined the computing landscape by enabling the emergence of disruptive, large-scale applications that would otherwise not have been possible with CPUs alone. *RTL simulators* play a key role in enabling the accelerated computing revolution: they are to hardware engineers what debuggers and runtime systems are to software engineers. Without RTL simulators, no hardware accelerator could be functionally designed. As accelerators increase in size and complexity, the hardware design industry will increasingly need faster RTL simulators to permit chip design in reasonable time.

Since the advent of multicore computers, parallelism is the preferred approach to improve software performance. RTL simulation seems to offer many opportunities to follow such a path: accelerators are written in hardware description languages that contain parallel constructs for describing independent hardware components that run in parallel and synchronize only at clock edges. Unfortunately, there is a mismatch between RTL simulation and today's multicore systems: tasks in RTL simulation tend to be very small in size, resulting in fine-grain parallelism. This fine-grain parallelism contrasts with coarse-grain parallel workloads for which modern multicore systems are built, which leads to simulator designs that can achieve only weak parallel performance scaling. This thesis argues that we need computing architectures that can achieve *strong scaling* to truly speed up RTL simulation through parallelism. A strong scaling architecture is one that can make effective use of additional cores without having to increase the total workload size. This enables even small or moderate size designs to exploit parallelism to run quickly.

This thesis contributes Manticore, a co-designed manycore architecture and compiler for RTL simulation that achieves strong parallel performance scaling. Manticore combines a bulk-synchronous parallel execution model with static scheduling to eliminate the runtime overheads of synchronization among hundreds of cores, simplify core design, and significantly increase the parallelism possible on a single chip. Our modest FPGA prototype of Manticore greatly increases parallel RTL simulation rate compared to a state-of-the-art software simulator running on top-of-the-line desktop and server x86 processors. The ideas underlying Manticore's design present a first step towards fast, scale-out RTL simulation.

Keywords: RTL simulation, parallelism, hardware acceleration, manycore architecture, FPGA

Résumé

La fin de la loi de Moore et de la mise à l'échelle de Dennard a entraîné une diminution des gains de performance pour les processeurs à usage général (CPU), ce qui a suscité un regain d'intérêt académique et commercial pour les accélérateurs matériels. Le matériel spécialisé a déjà redéfini le paysage informatique en permettant l'émergence d'applications à grande échelle qui n'auraient pas été possibles avec les CPUs seuls. Les *simulateurs RTL* jouent un rôle clé dans la révolution de l'informatique accélérée : ils sont aux ingénieurs matériels ce que les débogueurs et les systèmes d'exécution sont aux ingénieurs logiciels. Sans simulateurs RTL, aucun accélérateur matériel ne pourrait être conçu fonctionnellement. Au fur et à mesure que les accélérateurs augmentent en taille et en complexité, l'industrie de la conception matériel aura besoin de simulateurs RTL plus rapides pour permettre la conception de puces en un temps raisonnable.

Depuis l'avènement des ordinateurs multicœurs, le parallélisme est l'approche préférée pour améliorer les performances logicielles. La simulation RTL semble offrir de nombreuses opportunités pour suivre une telle voie : les accélérateurs sont décrits dans des langages de description matérielle qui contiennent des constructions parallèles pour décrire des composants matériels indépendants qui fonctionnent en parallèle et se synchronisent uniquement sur les flancs d'une horloge. Malheureusement, il existe un décalage entre la simulation RTL et les systèmes multicœurs d'aujourd'hui : les tâches en simulation RTL ont tendance à être très petites, ce qui entraîne un parallélisme à grain fin. Ce parallélisme à grain fin contraste avec les tâches parallèles à grain grossier pour lesquelles les systèmes multicœurs modernes sont construits, ce qui conduit à des implémentations de simulateurs qui ne peuvent atteindre qu'une mise à l'échelle faible de leur performance en utilisant du parallélisme. Cette thèse soutient que nous avons besoin d'architectures informatiques qui peuvent atteindre une *mise à l'échelle forte* pour accélérer réellement la simulation RTL à travers du parallélisme. Une architecture à mise à l'échelle forte est celle qui peut faire un usage efficace de cœurs supplémentaires sans avoir à augmenter la taille totale de la charge de travail. Cela permet même à des designs de petite ou moyenne taille d'exploiter le parallélisme pour s'exécuter rapidement.

Cette thèse contribue Manticore, une architecture et un compilateur manycore co-conçus pour la simulation RTL qui atteint une mise à l'échelle forte de sa performance parallèle. Manticore combine un modèle d'exécution parallèle synchrone en bloc (bulk-synchronous parallel) avec une planification statique pour éliminer les surcoûts de synchronisation entre des centaines de cœurs, simplifier la conception des cœurs, et augmenter considérablement le parallélisme possible sur une seule puce. Notre modeste prototype FPGA de Manticore augmente considérablement le

taux de simulation RTL parallèle par rapport à un simulateur logiciel à la pointe de la technologie fonctionnant sur des processeurs x86 de bureau et de serveur haut de gamme. Les idées sous-jacentes à la conception de Manticore représentent un premier pas vers une simulation RTL rapide et extensible.

Mots clefs : Simulateur RTL, parallélisme, accélération matériel, architecture manycore, FPGA

Contents

Acknowledgements **i**

Abstract **vii**

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**

1.1 The Stagnating Growth of CPU Performance 1
1.2 The Rise of Hardware Acceleration 1
1.3 The Bottleneck of Hardware Simulation 3
1.4 Thesis Contributions 5
1.5 Thesis Organization 6
1.5.1 Bibliographic Notes 7

I Hardware-Accelerated RTL Simulation **9**

2 Background **11**

2.1 Simulation Taxonomy 11
2.2 Circuit Representation 14
2.3 Simulator Implementation 14
2.3.1 Event-driven Simulators 14
2.3.2 Full-cycle Simulators 15
2.4 Summary 15

3 Manticore: An Architecture for Parallel RTL Simulation **17**

3.1 Serial Full-Cycle Simulation 17
3.2 Quantifying Parallelism in RTL Simulation 17
3.3 Parallel Full-Cycle Simulation 19
3.4 Bounding Parallel Simulation on General-Purpose CPUs 20
3.4.1 First Model 20
3.4.2 Second Model 22
3.4.3 Discussion 22
3.5 Are Other Architectures Suitable? 25

3.6 The Manticore Architecture 26
3.6.1 Key Insight 26
3.6.2 Architecture 26
3.6.3 Placing Manticore in the System Hierarchy 29
3.6.4 Interacting With Manticore 30
3.7 Summary 31

4 Manticore Microarchitecture and FPGA Implementation 33
4.1 Overview 33
4.2 FPGA Primitives 34
4.3 Core Design 36
4.3.1 Datapath Width 36
4.3.2 Allocating Resources to Manticore's Building Blocks 36
4.3.3 Instruction Set 38
4.3.4 Pipeline Implementation 39
4.4 Network-on-Chip Design 41
4.5 Global Stalling Mechanism 43
4.6 Off-chip Memory Access and Caching 44
4.7 Manticore Runtime, Bootloading, and Processor Execution 45
4.7.1 Bootloader Execution 45
4.7.2 Program Execution 47
4.8 Floorplanning 47
4.8.1 The Shell's Impact on Clock Frequency 48
4.8.2 High-level Floorplanning 49
4.8.3 Low-level Floorplanning 54
4.8.4 Floorplanning Results 55
4.9 Alternative Microarchitectures 55
4.9.1 Core Designs 55
4.9.2 NoC Designs 58
4.10 Summary 59

5 Manticore Compiler 61
5.1 Overview 61
5.1.1 Note on Program Optimizations 63
5.2 Frontend 63
5.2.1 Terminology 63
5.2.2 Compiling RTL to Netlist Assembly 63
5.3 Backend 64
5.3.1 Partitioning 66
5.3.2 Placement 69
5.3.3 Custom Function Synthesis 74
5.3.4 Scheduling 79
5.3.5 Register Allocation 79
5.4 Summary 80

6 Manticore Evaluation 81
6.1 Baseline Simulator . . . 81
6.2 Test Environment . . . 82
6.3 Benchmarks . . . 83
6.4 Performance Comparison . . . 84
6.4.1 Verilator . . . 85
6.4.2 Manticore . . . 86
6.5 Scaling Trends . . . 86
6.5.1 Performance . . . 86
6.5.2 Workload Distribution . . . 88
6.5.3 Instruction Duplication Overheads . . . 90
6.5.4 Where Is Time Spent? . . . 91
6.6 Compiler Optimizations . . . 93
6.6.1 Communication-Aware Partitioning . . . 93
6.6.2 Custom Instructions . . . 93
6.6.3 Placement . . . 95
6.6.4 Register Usage . . . 96
6.7 Compile Time . . . 97
6.8 Cost Analysis . . . 98
6.9 Global Stall . . . 100
6.10 Summary . . . 102

II Low-level Tools for FPGA Manipulation 105

7 A General Approach for Reverse-Engineering Xilinx Bitstream Formats 107
7.1 Motivation . . . 107
7.2 Device Structure . . . 110
7.3 Bitstream Structure . . . 111
7.3.1 Single-SLR Configuration . . . 111
7.3.2 Multi-SLR Configuration . . . 112
7.4 Locating a Frame by Address . . . 113
7.4.1 Frame Addressing Scheme . . . 113
7.4.2 Enumerating Frame Addresses . . . 115
7.5 Extracting Device Parameters . . . 116
7.5.1 Mapping Resource Columns to Major Column numbers . . . 117
7.5.2 Putting it Together: Forming a Partial Frame Address . . . 120
7.6 Extracting Architecture Parameters . . . 121
7.6.1 BRAM Format . . . 121
7.6.2 CLB Format . . . 121
7.6.3 Putting it Together: Forming a Full Frame Address . . . 123
7.7 Evaluation . . . 124
7.8 Discussion . . . 125
7.8.1 Minor Counts per Major Resource Column . . . 125
7.8.2 CLB Format . . . 127

7.8.3 BRAM Format 128
7.8.4 Frame Offsets and Physical Placement 130
7.8.5 SLR Similarities 130
7.8.6 Device Capacity 132
7.9 Summary 133

III Related and Future Work 135

8 Related Work 137
8.1 Software Simulators, FPGA Prototypes, and Emulators 137
8.2 Custom Functions 138
8.3 Sequential RTL Simulation 138
8.4 Parallel RTL Simulation Using CPUs 139
8.5 Parallel RTL Simulation Using GPUs 140
8.6 Deterministic Acceleration 140

9 Future Work 141
9.1 Language Support 141
9.2 Multiple Clock Domains 141
9.3 Waveform Debugging 142
9.4 Physical Implementation 143
9.5 Multi-FPGA Simulation 143
9.6 Timing-Accurate Simulation 144
9.7 Using Accelerators Designed for Other Problem Domains 144

10 Conclusion 145

Bibliography 157

Curriculum Vitae 159

List of Figures

1.1 Improvement in CPU performance over 40 years 2
1.2 Publicly announced machine learning accelerators as of 2022 4

2.1 Hardware system modeling graph 12
2.2 An example single-clock netlist 14

3.1 Cycle-accurate simulation of a netlist 18
3.2 Bulk-synchronous parallel RTL simulation 20
3.3 BSP strong scaling experiment on multicore shared-memory processors 21
3.4 Idealized simulator performance 23
3.5 Manticore high-level architecture 27
3.6 Global stall 28

4.1 Manticore block diagram 34
4.2 U200 FPGA 35
4.3 Core pipeline diagram 39
4.4 Custom function unit 41
4.5 Uni-directional NoC layout 42
4.6 Core NoC ingress datapath 46
4.7 Regular placement of a 25×10 Manticore grid 50
4.8 Irregular split grid floorplan 52
4.9 Decoupling core placement from NoC switch placement 53
4.10 Enforcing relative placement of BRAMs and URAMs in cores 54
4.11 Automatic vs. guided floorplanning 56

5.1 Compilation overview 62
5.2 Extracting parallelism through partitioning 68
5.3 Placing processes on cores 69
5.4 Optimal process-to-core assignment MILP formulation 71
5.5 Sensitivity of link weights to minor changes in process-to-core assignment . . . 73
5.6 Synthesizing custom functions from Verilog code 75
5.7 Cuts and fanout-free cones (FFCs) in a logic subgraph 76
5.8 Identifying custom functions in a netlist DAG 78
5.9 Custom function synthesis MILP formulation 78

6.1 Verilator's multi-threaded compilation flow 82

6.2 Verilator and Manticore simulation rate . . . 85
6.3 Verilator and Manticore simulation scaling . . . 87
6.4 Workload imbalance between cores . . . 89
6.5 Instruction duplication overhead when scaling Manticore designs . . . 91
6.6 Manticore aggregate stall breakdown . . . 92
6.7 Communication-oblivious vs. communication-aware partitioning . . . 94
6.8 Custom functions' contribution to performance . . . 95
6.9 Ideal NoC performance . . . 96
6.10 Register file lifetime analysis on a 15×15 Manticore grid . . . 97
6.11 Breakdown of compilation time . . . 99
6.12 Manticore's cache effectiveness . . . 101

7.1 Floorplan of an Alveo U50 datacenter FPGA . . . 109
7.2 The frame address format in UltraScale and UltraScale+ devices . . . 112
7.3 The SLR frame indexing scheme . . . 114
7.4 The process for mapping a bit in a LUT's `INIT` property to a bit in the bitstream 117
7.5 Logic location file excerpt . . . 118
7.6 Zoomed-in region in cyan from Figure 7.1 . . . 119
7.7 CLB and interconnect noise in a bitstream . . . 120
7.8 LUT and flip-flop `INIT` bits in UltraScale and UltraScale+ devices . . . 127
7.9 BRAM `INIT` bits in UltraScale and UltraScale+ devices . . . 129
7.10 Relationship between BEL frame offsets and vertical placement in a clock region 131
7.11 Clock region X0Y0 in the Alveo U250 and U280 datacenter-grade FPGAs . . . 132
7.12 Floorplan of a `xcau10p` device . . . 133

9.1 Scheduling clock activation in a full-cycle simulator . . . 142

List of Tables

3.1 Benchmark RTL circuit netlist DAG statistics 19

4.1 Alveo U200 critical resource analysis 37
4.2 Manticore implementation results 48
4.3 Resources needed to implement 1024-entry multi-ported register file 57

6.1 Hardware platforms 83
6.2 Verilator and Manticore simulation performance 84
6.3 Compile times across all platforms 98
6.4 Hourly cost of Microsoft Azure instances 99
6.5 Simulation cost using Microsoft Azure prices 100

7.1 Frame addresses and offsets of `SLICE_X136Y247/A6LUT` on a U50 FPGA 123
7.2 Summary of devices on which Bitfiltrator was tested 124
7.3 Bitstream configuration word breakdown 126
7.4 Devices grouped by SLR frame addresses 132

1 Introduction

CPU performance growth has stagnated; demand for increased computing power has not. The imbalance between our CPUs' processing power and the computational requirements of today's applications has led to massive demand for *specialized hardware*. These devices, purpose-built for specific applications, exploit the inherent parallelism and unique characteristics of their target workloads to outperform CPUs in terms of speed and efficiency. With increased interest in specialized hardware comes a strong demand for fast ways to design and validate them.

1.1 The Stagnating Growth of CPU Performance

Driven by Moore's Law, the number of transistors in computer processors has doubled every 18–24 months since the 1970s, resulting in modern commodity chips containing upwards of 50 billion transistors. This exponential increase in transistors would have been meaningless if not accompanied by beneficial improvements. Fortunately, a key improvement did occur: CPU performance has increased by at least five orders of magnitude over this period! This growth in CPU performance owes much to the favorable circumstances enabled by Moore's Law and Dennard scaling.

CPU performance, however, has not increased at a constant rate throughout history. Figure 1.1 plots the annual growth rate of CPU performance since 1978 and shows that growth in CPU performance has *decelerated* post-2003 with the end of Dennard scaling. CPUs are now power-limited and clock frequencies increase slowly, signaling that technology scaling no longer contributes to performance growth to the same extent as before. There also is a limit to the amount of instruction-level parallelism (ILP) that can be extracted from sequential programs [109]. The net result is that—as of 2018—CPU performance growth has largely flattened out, sustaining only 3.5% year-on-year improvement.

1.2 The Rise of Hardware Acceleration

Riding the tides of technology scaling is no longer an effective method to increase CPU performance, so the computing industry must look for alternatives to sustain growth in computing

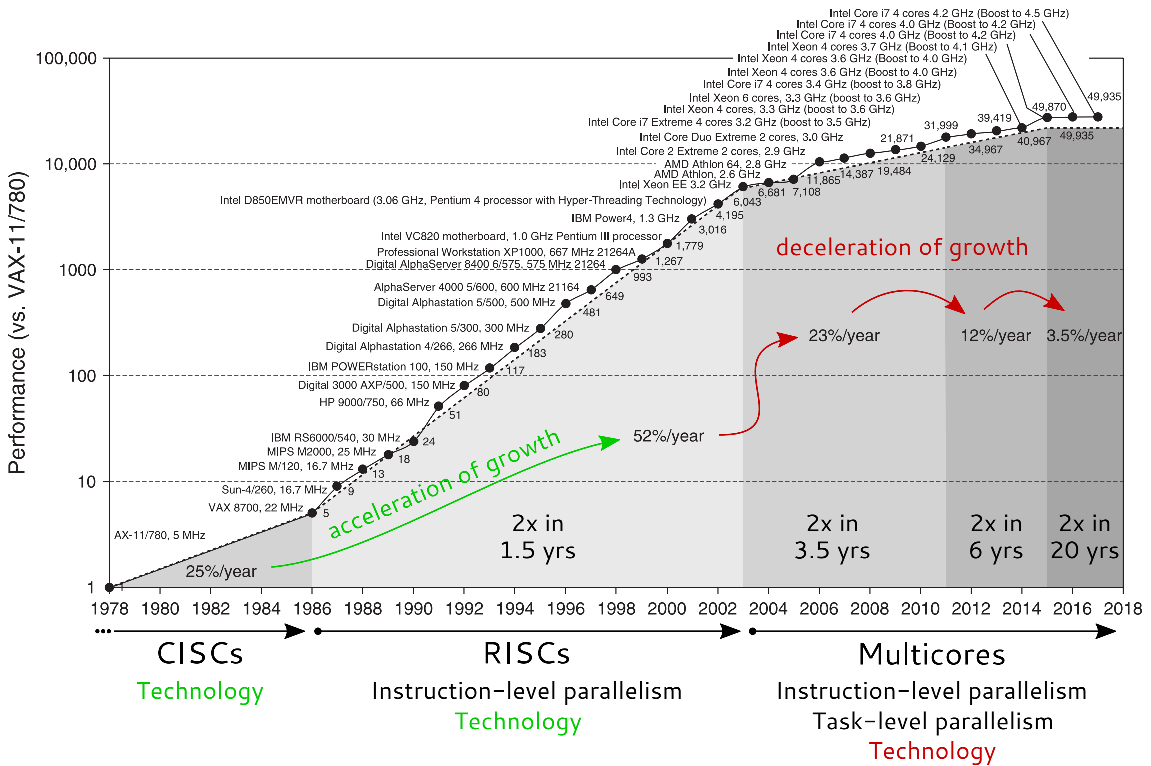


Figure 1.1 – Improvement in CPU performance over 40 years. The performance of machines in the CISC era, RISC era, and multicore era experienced different annual growth rates. The text underneath each era describes the main driver of performance growth during the period. Figure reproduced and adapted from Hennessy and Patterson [44].

power. *Hardware acceleration* appears to be a viable means to this end. An accelerator is a specialized piece of hardware that is purpose-built to solve a particular task. By eschewing the one-size-fits-all approach of a *general-purpose* CPU, an accelerator can instead implement *domain-specific* optimizations to improve performance and efficiency.

CPUs are built to exhibit good performance for diverse workloads, at the expense of not excelling at any specific one. They do so using von Neumann architectures with standard functional units that operate on memory-level words. CPUs generally attempt to improve performance by exploiting more ILP. By contrast, accelerators must exhibit exceptional performance and efficiency for only a specific class of workloads. They do so using an architecture chosen for their problem domain, typically with custom datapaths and functional units that facilitate data movement and computation on application-level objects. Accelerators improve performance by exploiting multiple forms of parallelism (data, memory, stream, pipeline, fine-grained, model, task, etc.) efficiently.

There exists a diverse set of accelerator architectures. We briefly outline the most common ones:

Dataflow Computing paradigm in which problems are expressed as functional units and the data flowing between them. Dataflow machines are typically dynamic: functional units can execute as soon as data is available (rather than adhering to a fixed instruction sequence).

Spatial An accelerator featuring an array of relatively simple ALU-like processing elements (PEs) capable of direct communication with one another through a Network on a Chip, bus, or inter-PE connection [29].

Systolic Architectures wherein computation is performed by a 1D or 2D array of arithmetic units controlled in lockstep. They are a special case of spatial accelerators with two restrictions: (1) PEs perform a fixed function on the input data, and (2) data flows between PEs in a fixed direction. Systolic architectures produce partial results in a wave-like fashion [52].

Vector Architectures designed to perform mathematical operations on vectors and matrices in a highly parallelized manner [94]. Vector processors can load and store data from memory both in contiguous and strided patterns. A vector architecture is optimized to perform the same operation on multiple elements of a vector or matrix simultaneously, typically by using a single instruction.

Manycore Accelerators that feature many hundreds or even thousands of physical cores, often arranged in groups of ten or more multiprocessors. Manycores can be considered an evolutionary step beyond their multicore counterparts, exhibiting even greater parallelism. To achieve such levels of parallelism, individual cores are relatively simple (in-order, etc) to enable more dense packing onto a single chip [35].

GPU A special case of manycore architecture where each core is a multithreaded vector processor [71].

Accelerators come in multiple shapes (FPGA, GPU, ASIC) and exist for a variety of applications: machine learning [6, 29, 51, 52, 77], video processing [3, 86], networking [72], storage [62], bioinformatics [97], etc. Machine learning (ML), in particular, has aroused massive commercial interest. Given the exponential increase in the size and complexity of ML models, the computing industry has poured large sums into the development of hardware accelerators for ML workloads. Figure 1.2 shows the sheer number of ML accelerators that have been publicly announced as of 2022 and their location in the performance-power design space.

In summary, interest in hardware accelerators has grown as the computing industry seeks alternative ways to boost computing performance and efficiency in the face of the diminishing returns of Moore's Law and Dennard scaling. Hardware accelerators will likely continue to spread to other fields as they become increasingly dependent on computation.

1.3 The Bottleneck of Hardware Simulation

During the development process, both software and hardware artifacts must undergo various stages of implementation, debugging, functional verification, and performance validation. While software artifacts can simply run their code on a machine to do so, hardware artifacts cannot "run" a workload as they are yet to be physically manufactured. As a result, hardware designs are *simulated* on a computer using specialized software tools that model the behavior of the design.

Hardware simulation is necessary to ensure that a design meets its intended functionality and performance specifications before it is committed to silicon. Unlike software, hardware bugs—whether functional or performance-related—generally cannot be patched after a system has been

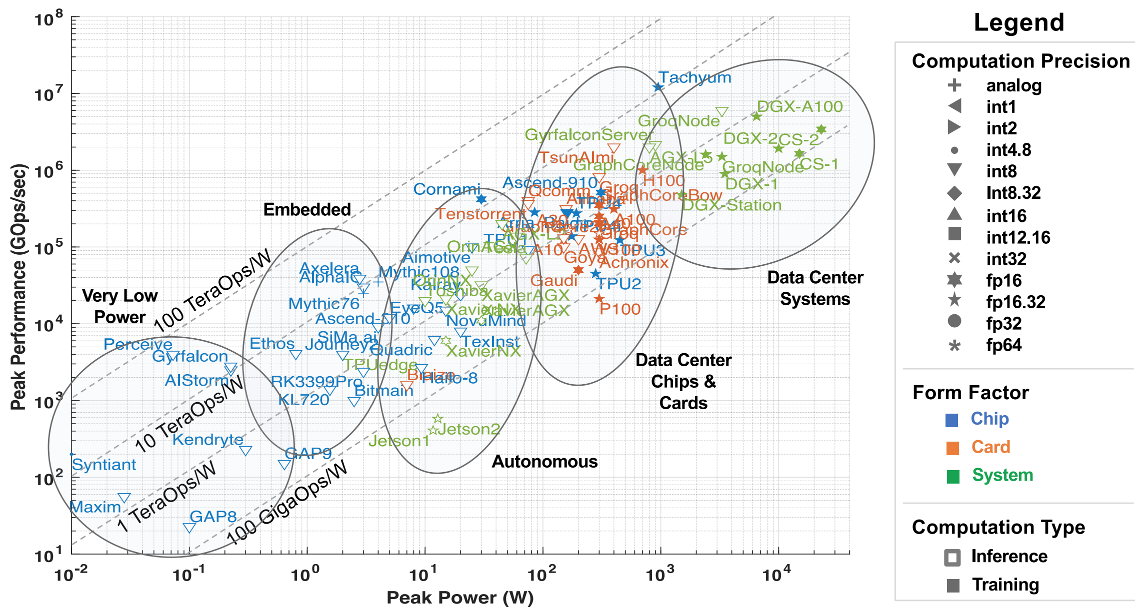


Figure 1.2 – Publicly announced machine learning accelerators as of 2022. Figure reproduced and adapted from Reuther et al. [79].

built: Hardware bugs which escape early design stages can be fixed at a certain cost, but those which are detected only after shipping are prohibitively expensive to address [30].

As demand for faster and more efficient processing continues to grow, hardware accelerators will become larger and more complex, which entails more involved simulation. Ideally we want RTL simulators that can achieve a simulation rate of approximately 1 MHz; much slower than hardware, but fast enough to avoid days worth of simulation time for large designs. Fast simulators enable engineers to perform more design runs per day, which in turn permits finding higher-performing designs, increasing coverage for verification, and identifying bugs more quickly.

Designers, however, face a dilemma. Software RTL simulators offer excellent visibility into hardware internals and can compile large hardware designs in a few minutes, but are very slow at runtime: the fastest software RTL simulators can simulate designs at a rate of only 1 kHz to 1000 kHz, i.e., more than three orders of magnitude slower than the physical hardware. By contrast, prototyping a hardware design on an FPGA permits simulation rates in the range of 10 MHz to 100 MHz, but offers only limited visibility into design state and takes a very long time to compile (hours- or days-long place-and-route tools are needed to map hardware to the FPGA). Slow runtime or compile time is a bottleneck when performing extensive design space exploration or testing.

Since the advent of multicore computers, *parallelism* is the preferred approach to improve software performance. RTL simulation seems to offer many opportunities to follow such a path: accelerators are written in hardware description languages that contain parallel constructs for describing independent hardware components that run in parallel and synchronize only at clock edges. RTL designs therefore comprise many independent tasks, which suggests that RTL

simulation may be a good fit for execution on multicore machines, built for task-level parallelism. However, tasks in RTL simulation tend to be very small in size: they consist of the combinational gates between two flip-flops or memories, and the number of gates is typically no more than a few tens to a few hundreds. This results in *fine-grain* parallelism. By contrast, shared-memory multicore machines are designed for *coarse-grain* parallelism as the overhead of coordinating fine-grain tasks across multiple cores can be prohibitively high, resulting in diminishing returns as more cores are used.

Parallel software RTL simulators work around this mismatch by grouping multiple tasks together to coarsen the granularity of computation, reducing the overhead of coordination between cores and resulting in parallel speedup. However, since the cost of coordination among cores increases as more cores are used, the coarsened tasks must also increase in size to keep coordination overheads constant, which is only possible if the total workload size increases. This suggests that multicore machines are capable of speeding up simulation only through *weak scaling*: they can use more cores to maintain the same parallel speedup for increasing problem sizes, but cannot increase the parallel speedup of a fixed-sized problem past some threshold without decreasing performance.

If we truly want to use increased parallelism to improve RTL simulation performance, we need computing architectures that can achieve *strong scaling*. A strong scaling architecture is one that can make effective use of additional cores *without* having to increase the total workload size. In such an architecture, adding more cores reduces each core's workload, leading to an increased simulation rate. A strong scaling architecture, with enough cores and balanced per-core workloads, could enable RTL simulation rates to approach the 1 MHz range.

1.4 Thesis Contributions

This thesis proposes Manticore: a strong scaling, manycore accelerator for efficient, parallel RTL simulation.

Manticore combines a bulk-synchronous parallel execution model with static scheduling (i.e., "static BSP") to eliminate the runtime overheads of synchronization among hundreds of cores. Since the scheduled synchronization occurs without overhead, fine-grain interactions among cores are efficient. Device-wide static scheduling also allows us to simplify the Manticore processors, significantly increasing the parallelism possible on a single chip.

Like MIT's Raw machine [108], Manticore relies entirely on its compiler to schedule resources and communication. Manticore's compiler accepts single-clock RTL designs and generates binary code for a Manticore accelerator. Compilation time is comparable to software compilers, offering software development-like turnaround and fast simulation rate, especially useful for hours- to days-long simulations.

We prototyped Manticore on an FPGA, and it outperforms Verilator [92], the fastest open-source RTL simulator, running on top-of-the-line multicore general-purpose processors despite running at a fraction of their clock speed. Hardware-accelerated simulation offers a way out of the

dilemma posed above by optimizing time-to-result. Small experiments and tests can run on a software simulator with rapid turnaround. More extensive experiments and tests can run on Manticore, with slightly slower compile times, but much faster execution. And hardware prototypes can be reserved for full-system simulation, operating system bring-up, and software development.

The main contributions of this thesis are:

- An application of the static BSP execution model to RTL simulation,
- The Manticore architecture that employs fine-grain parallelism to simulate RTL,
- A compiler that finds parallelism in RTL code and statically schedules it to run effectively on Manticore,
- A high-performance FPGA prototype of Manticore, and
- An evaluation comparing and analyzing the performance of Manticore against state-of-the-art software RTL simulation.

1.5 Thesis Organization

This thesis is organized in two orthogonal, but related parts.

Part I presents the main contribution of this thesis: the design, implementation, and evaluation of the Manticore manycore accelerator for RTL simulation. Manticore is based on work to appear in ASPLOS'24.

- Chapter 2 presents background information on hardware simulation and provides a taxonomy for the different granularities at which hardware circuits are simulated. We then frame the specific class of simulators addressed by this thesis: full-cycle, cycle-accurate RTL simulators.
- Chapter 3 quantitatively studies the amount and type of parallelism available in RTL simulation. We then demonstrate that shared-memory multicore machines are fundamentally limited in their ability to effectively exploit the fine-grain parallelism available in RTL simulation beyond few tens of cores. We also explain why the next most common parallel architecture available today—the GPU—is not a viable platform for accelerated single-instance RTL simulation. We end by presenting the key ideas underlying Manticore, our proposed manycore architecture for scalable parallel RTL simulation.
- Chapter 4 describes Manticore's microarchitecture and FPGA implementation on a large Xilinx UltraScale+ device, in particular the design choices we made to permit a dense and high clock frequency implementation.
- Chapter 5 introduces Manticore's compiler, which uses the hardware's determinism to schedule RTL code to instructions that Manticore's hundreds of cores can efficiently execute. Manticore's deterministic hardware lacks interlocks and buffering, so it relies entirely on its compiler to parallelize the input netlist, schedule instructions within each core to avoid data hazards, and schedule messages between cores to avoid structural hazards on Manticore's NoC.

- Chapter 6 evaluates Manticore's performance, cost, compile time, and design decisions. We compare Manticore's performance against Verilator, the fastest full-cycle RTL simulator, running on three high-end desktop and server x86 processors. We analyze Manticore's cost in a cloud environment using the closest comparable FPGA and high-performance processors. We separately evaluate each of Manticore's design decisions (communication-aware partitioning, custom functions, program placement, register capacity) and their contribution to overall performance.

Part II presents a secondary contribution of this thesis: the Bitfiltrator FPGA bitstream manipulation tool. Bitfiltrator is based on work published in FPL'22.

- Chapter 7 describes how to reverse-engineer parts of a Xilinx FPGA's bitstream to support alternative programming workflows. Bitfiltrator was originally conducted in the context of the Manticore project. The goal was to rapidly program parts of Manticore's functional units at the FPGA bitstream-level so we could omit Manticore's programming circuitry from its design and simplify timing closure at high clock speeds. However, we did not end up using Bitfiltrator in the final Manticore prototype, and so we decided to spin it off as educational material at an FPGA conference to teach others how to reverse-engineer parts of an FPGA's bitstream as this information was lacking in the literature. At the time when this project was conducted, Bitfiltrator was the only tool capable of editing the bitstream of large, multi-die Xilinx FPGAs.

1.5.1 Bibliographic Notes

This thesis expands upon work conducted during the final two years of my PhD, resulting in the following two publications:

Manticore: Hardware-Accelerated RTL Simulation with Static Bulk-Synchronous Parallelism
Mahyar Emami, Sahand Kashani* (*equal contribution)*
Keisuke Kamahori, Mohammad Sepehr Pourghannad, Ritik Raj, James R. Larus
In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24).

Bitfiltrator: A General Approach for Reverse-Engineering Xilinx Bitstream Formats
Sahand Kashani, Mahyar Emami, James R. Larus
In Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications (FPL'22).

Manticore was a massive project and was largely the result of joint work between myself and my colleague Mahyar Emami. This thesis presents Manticore as a whole for completeness so that its description is coherent. However, I will dive into the details of only the parts to which I made major contributions.

Hardware credit

I proposed the overarching idea of using static scheduling in a manycore architecture to remove the overheads of runtime synchronization in RTL simulation. I evaluated the feasibility of using clock gating as a global stalling mechanism on a large, multi-die FPGA, which would permit a physical implementation of Manticore. Mahyar and I sketched the high-level details of many of Manticore's core modules, which Mahyar then used to implement a first version of Manticore while focussing on correctness, not on speed. I sketched a detailed implementation of an improved 500 MHz core, which an intern (Keisuke Kamahori) helped implement. I then performed a cross-sectional redesign of Manticore's cores, switches, and control structures to permit scaling its architecture to large grid sizes, while remaining clocked near our desired 500 MHz target.

Compiler credit

I developed a first version of the compiler's Verilog frontend in Yosys' experimental Python bindings, which Mahyar later ported and expanded into Yosys' C++ codebase. Mahyar developed the backend compiler's overarching framework, on top of which we both contributed various standard optimization passes (constant folding, dead code elimination, etc.). The partitioner, scheduler, register allocator, and bootloading code are entirely Mahyar's work. I focused on studying the impact of program placement strategies and custom function synthesis on Manticore's performance.

Part I

Hardware-Accelerated RTL Simulation

2 Background

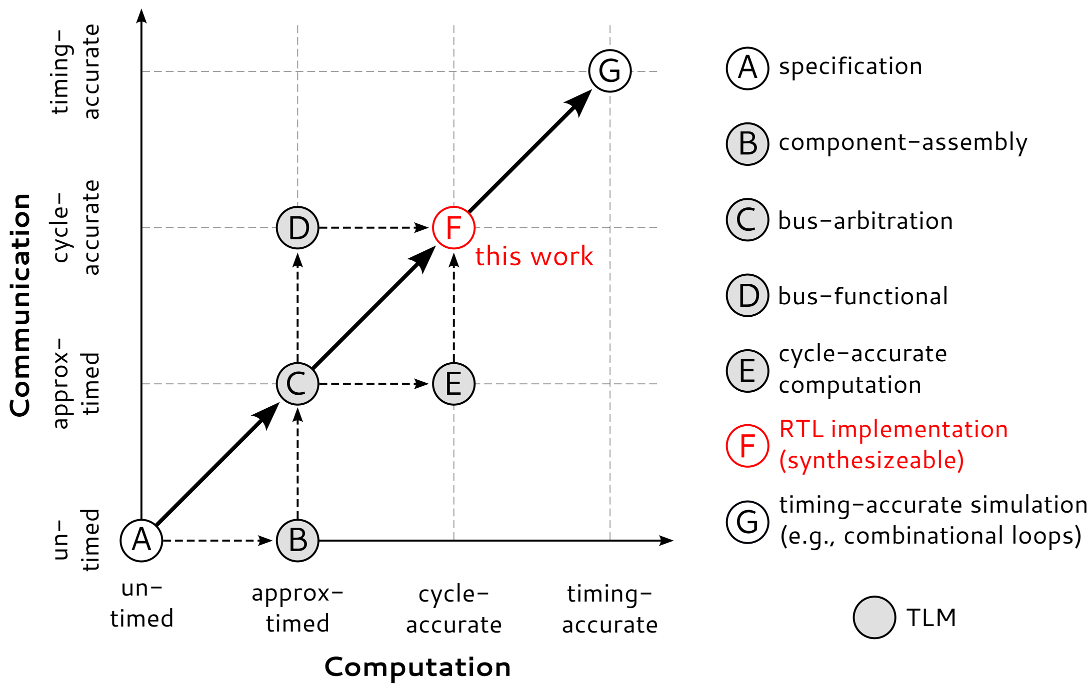
Hardware accelerators require multiple iterations of implementation, debugging, functional verification, performance validation, and design space exploration. Given the high cost of taping out chips, a significant fraction of an engineer's time is spent performing detailed hardware *simulation* [38, 39]. A simulator's turnaround time and throughput can directly affect designer productivity and product quality: faster simulators enable engineers to perform more design runs per day, which in turn permits identifying bugs more quickly, increasing coverage for verification, and finding higher-performing designs.

This thesis is about using *parallelism* to speed up the simulation of hardware accelerators. We defer a discussion of parallel simulation to Chapter 3. This chapter provides a taxonomy for the different granularities at which hardware circuits are simulated. We then frame the specific class of simulators addressed by this thesis.

2.1 Simulation Taxonomy

Hardware circuits can be simulated using a variety of models: (1) un-timed, (2) transaction-level, (3) cycle-accurate, and (4) timing-accurate models. Each model provides increasing simulation accuracy and is more costly to run:

- An *un-timed* model is a functional description of a design and does not contain any implementation details; it is purely algorithmic.
- A *transaction-level* model (TLM) is an abstract model suitable for early architecture exploration and performance analysis, without requiring detailed implementation. This abstraction reduces the complexity of the design process, and the increased speed of simulation allows for more extensive design space exploration. The execution time of a transactionally-modeled hardware design is *estimated* at the system level. There are multiple types of TLMs depending on the accuracy with which they model a hardware design's compute and communication units.
- A *cycle-accurate* model contains the implementation details of a *synchronous* system (i.e., a system that has at least one clock signal that synchronizes state elements in the circuit). It allows designers to *measure* execution time with cycle-level accuracy.



System Model	Computation		Communication	
	Accuracy	**Interface**	**Accuracy**	**Interface**
(A) Specification	none	no HW	none	variable
(B) Component-assembly	approx.	abstract	none	message-passing
(C) Bus-arbitration	approx.	abstract	approx.	abstract bus
(D) Bus-functional	approx.	abstract	cycle	detailed bus
(E) Cycle-accurate compute	cycle	pin	approx.	abstract bus
(F) Implementation (synthesizeable RTL)	cycle	pin	cycle	wire
(G) Timing-accurate	time	pin	time	wire

Figure 2.1 – Hardware system modeling graph. Thick solid arrows represent the traditional hardware design flow: designers start from an un-timed specification (A), devise an abstract system architecture (C), and finally refine computation and communication in the form of a cycle-accurate RTL implementation (F). If a design contains timing-dependent behavior (e.g., combinational loops), then a finer-grained timing-accurate RTL model (G) is needed to simulate it. The dashed arrows represent different intermediate models that can be used to reach those in the traditional design flow. Gray circles represent transaction-level models (TLMs), i.e., models that approximate some aspect of a system's implementation. Reproduced and adapted from Cai and Gajski [20].

- A *timing-accurate* model contains gate-level delays and allows simulating hardware circuits that have timing-dependent behavior (e.g., ring oscillators, etc.).

The system modeling graph and the table in Figure 2.1 characterize multiple hardware models by the accuracy with which they model elapsed time in a hardware design's compute and communication units:

- A specification model (A) contains an un-timed algorithmic description of a system as a collection of processes that communicate through global variables.
- A component-assembly model (B) describes a system's architecture as a collection of components that are individually responsible for either computation or communication. A compute unit's execution time is approximated using *wait* statements, whereas communication is un-timed and is performed through message-passing channels (no specific bus protocol is implemented; the channels model only data transfers and synchronization between compute units).
- A bus-arbitration model (C) describes how different components on a shared bus access the bus and communicate with each other. It includes high-level information about the arbitration mechanism used to determine which component has access to the bus at any given time. Arbitration entails knowing the order at which different accesses occur on a bus, and so both computation and communication are approximately-timed (using *wait* statements).
- A bus-functional model (D) describes a communication protocol's pin-level interface and control sequences (i.e., computation is approximate, but communication is cycle-accurate). Control sequences may be described using time delays (e.g., signal `write` precedes signal `data` by 50 ns, etc.), or is directly described using cycle-level behavior (which implies the existence of a clock signal). If the protocol is described using time delays, then the protocol needs to be refined into a cycle-level model to be implemented.
- A cycle-accurate computation model (E) is the compute-equivalent of a bus-functional model: computation is cycle-accurate, but communication is approximate. A cycle-accurate computation model is useful for obtaining detailed performance metrics for each compute unit individually.
- An implementation model (F) is a *synthesizeable* cycle-accurate description of an entire system (computation and communication). Cycle-accurate simulation captures value changes only at clock edges; there is no notion of time. A cycle-accurate simulator cannot model structures like combinational feedback loops as it does not see signal changes between clock edges. Fortunately most digital systems are *synchronous* and do not contain delay-sensitive logic, which greatly simplifies hardware design and testing.
- A timing-accurate model (G) simulates a hardware design by timestamping value changes to model gate delays accurately. It is required when asynchronous events must be modeled, i.e., when the notion of a "cycle" is undefined or when what happens in a cycle is unknown ahead of time. Examples include (1) systems without a clock, (2) systems with combinational loops (e.g., phase-locked loops, ring oscillators, etc.), and (3) systems that can dynamically adjust their clock frequency (e.g., with dynamic voltage frequency scaling). Timing-accurate simulation can be done cycle-accurately if combinational feedback loops are designed such that they always converge.

We focus on *synchronous* hardware systems in the rest of this thesis as they encompass the vast majority of hardware designs (processors, accelerators, etc.). A cycle-accurate simulation (F) of such systems provides the highest level of accuracy (beyond what a TLM can provide).

Before we dive into how clocked circuits are simulated, we first review how they are represented.

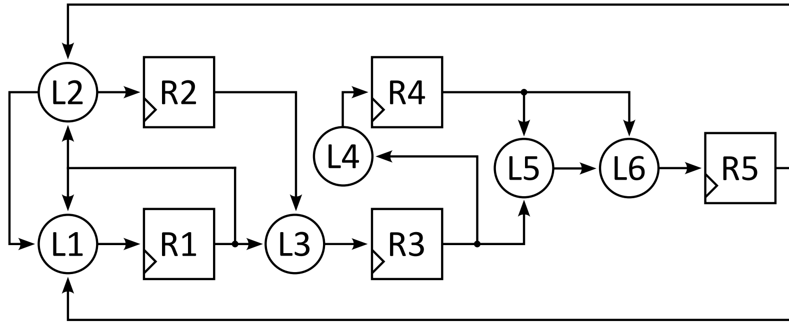


Figure 2.2 – An example single-clock netlist. Circles represent gates and rectangles represent registers.

2.2 Circuit Representation

Circuits are hierarchically described in hardware description languages (HDLs) such as VHDL or Verilog, i.e., languages that support parallel constructs for describing parallel hardware components that run independently and synchronize only at clock edges. Circuit simulation is performed on HDL sources even if the circuit is generated from a higher-level un-timed language as HDL is the single common denominator at which multiple hardware systems can be stitched together.

HDLs represent hardware circuits at a level of abstraction known as register-transfer level (RTL). RTL is just a fancy name for a set of registers and a set of transfer functions that connect them [82]. The registers represent circuit state and the transfer functions are stateless combinational logic that operate on registers and wire them together. HDLs therefore represent circuits in the form of a *netlist*: a directed graph whose nodes are circuit cells (gates, registers, and memory banks) and whose edges are the wires connecting them. Figure 2.2 shows an example netlist in which rectangles represent registers and circles represent gates.

2.3 Simulator Implementation

Simulating an RTL circuit in software is orders of magnitude slower than running the actual hardware since a CPU needs to use instructions to emulate up to millions of gates at each clock cycle. This is not an issue for small designs, but can lead to hours-long simulations for large designs if they must be simulated over billions of clock cycles. So how can we design simulators for fast execution in software?

2.3.1 Event-driven Simulators

Signals in many digital systems exhibit low activity factors and rarely change between clock cycles [24, 89]. This suggests that most of the simulation's results can be reused to save computation. An *event-driven* simulator propagates signal updates as events to its consumers, which are then *dynamically* scheduled for evaluation.

In principle event-driven simulators should yield significant savings on a traditional CPU as they avoid needless re-evaluation of unchanging circuit elements. In practice, monitoring a centralized event queue to trigger execution of circuit partitions greatly outweighs its benefits [111]. Designers want circuits that run at high clock frequencies, which constrains the number of gates between clock edges. A realistic RTL design therefore comprises many *tiny*, independent computation tasks which require no more than tens of instructions to emulate in software. Many more instructions may be required to schedule each task's computation depending on the event queue's implementation.

2.3.2 Full-cycle Simulators

While event-driven simulators evaluate only the signals that have changed in the previous cycle, a *full-cycle* simulator evaluates the *entire* circuit at each clock cycle. A single *static* schedule is used to ensure all signals are computed in the correct order, which eliminates the dynamic overheads of event-driven simulation. As a result, CPUs exhibit significantly higher simulation rates when running a full-cycle simulator compared to an event-driven one, despite the unnecessary extra work [11].

Side Note: Hybrid Simulators

Event-driven and full-cycle simulators are two extremes of the simulator implementation spectrum. A hybrid simulator blends properties of both implementations to reduce the runtime overheads of event-driven simulation, while retaining its benefit of reducing work in the presence of low activity factors. Hybrid simulators can improve *single-threaded* simulation performance by over an order of magnitude [11].

This thesis is about using *parallelism* to improve simulation performance, which is orthogonal to the techniques devised by hybrid simulators for improving single-thread simulation performance. The rest of this thesis uses a "pure" full-cycle simulator as the starting point for parallel simulation as it provides a good basis for understanding the source of parallelism in RTL simulation and how to effectively exploit it.

2.4 Summary

Simulation is an essential part in the hardware design workflow. The vast majority of real hardware circuits are *synchronous* and are described with cycle-level accuracy in hardware description languages. A full-cycle simulator executes the entire body of a circuit at each cycle to avoid the overheads of fine-grained event-driven simulation.

This concludes the high-level background on simulation. The next chapter covers how parallelism can be used to speed up simulation.

3 Manticore: An Architecture for Parallel RTL Simulation

Chapter 2 described full-cycle, cycle-accurate simulators as being fast, but they are still orders of magnitude slower than hardware. We must explore other dimensions in the design space to speed up RTL simulation. Since the advent of multicore machines, parallelism is the preferred approach to improve software performance, and so this chapter introduces parallel, full-cycle RTL simulation.

We start by presenting serial full-cycle RTL simulation. Next, we quantify the amount and type of parallelism available in RTL simulation. We then demonstrate that shared-memory multicore machines are fundamentally limited in their ability to effectively exploit the type of parallelism available in RTL simulation beyond few tens of cores. Finally, we conclude by presenting *Manticore*, our proposed architecture for scalable parallel RTL simulation.

3.1 Serial Full-Cycle Simulation

Figure 3.1 shows the two-stage process used to evaluate a netlist graph.

First, the netlist graph (top) is made acyclic by splitting the state nodes (e.g., registers) into a *current* and *next* value. This results in a directed acyclic graph (DAG), where current and next values are denoted by − and +, respectively (bottom).

Second, the netlist DAG is executed while respecting its precedence relations. A simulated cycle concludes when all next register values are computed using the current register values. The current values are then updated from the newly computed ones, and the process repeats.

3.2 Quantifying Parallelism in RTL Simulation

We want to exploit parallelism to speed up RTL simulation, so it is important to start by quantifying how much parallelism exists in real circuits. If there is little parallelism, then parallelism alone would not lead to good speedups; aggressive sequential execution techniques (speculation, etc.) would be better.

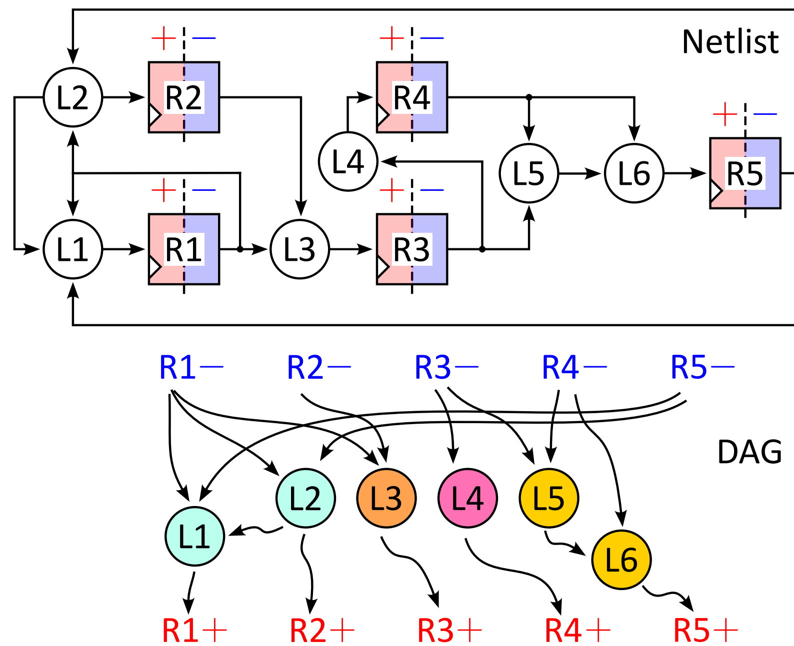


Figure 3.1 – Cycle-accurate simulation of a netlist. Circles represent gates and rectangles represent registers. Simulating a netlist involves splitting registers into a *current* and *next* value denoted by − and +, respectively (top). The resulting split circuit is then represented as a DAG (bottom) where *current* values are inputs and *next* values are outputs. Executing the DAG simulates the operations that occur in one clock cycle. Colored circles in the DAG represent independent paths that can be computed in parallel.

Notice that a netlist DAG fully expresses the inherent parallelism of an RTL circuit as an interpreter can simulate independent paths through the graph in parallel. Figure 3.1 highlights the vertices along a given path with the same color and shows that even this tiny RTL circuit exhibits 4-way parallelism.

Table 3.1 extends this analysis to nine small- and medium-sized RTL benchmarks (described in Section 6.3). It reports three metrics for each benchmark: (1) the total number of vertices in its netlist DAG, (2) the depth of the DAG, and (3) the width of the DAG. The number of vertices in the DAG is a measurement of how much total work is needed to simulate the entire circuit at each clock cycle, while the depth is a measure of the largest serial portion of the circuit (i.e., its critical path). The available parallelism in the DAG depends on how its vertices are scheduled. We estimate this parallelism after ASAP scheduling by computing the minimum, average, median, and maximum DAG widths. We observe that the number of vertices in a netlist DAG greatly outnumbers its depth: netlist DAGs are *shallow* and *wide*, and so have ample parallelism. The only outlier is `jpeg`, where the average width of the DAG is at least an order of magnitude lower than other benchmarks. The `jpeg` benchmark is therefore likely to benefit little from parallelism.

In summary, RTL simulation offers many opportunities for parallelism as circuits are composed of exclusively of parallel hardware components that run independently, i.e., they exhibit abundant task-based parallelism. Furthermore, this parallelism is made explicit by circuit designers directly in HDLs, so no parallelism needs to be extracted.

Benchmark	Size	Depth	Width			
			min	avg	med	max
jpeg	3837	188	2	20	4	600
blur	7653	23	3	319	197	1332
bc	16612	19	2	831	367	4861
mc	48161	24	32	1926	395	8275
mm	66328	23	225	2764	1956	8488
rv32r	71094	84	16	836	337	6308
noc	99033	71	16	1375	120	11423
cgra	105500	47	49	2198	1078	10125
vta	117062	143	1	812	4	21836

Table 3.1 – Benchmark RTL circuit netlist DAG statistics. A vertex corresponds to an *arbitrary-width* operation as specified in the original Verilog. The size of the DAG represents the total amount of work needed to simulate the entire circuit at each clock cycle. The depth of the DAG quantifies the largest serial portion of the circuit. We estimate the amount of parallelism in the DAG by computing its minimum, average, median, and maximum width through ASAP scheduling. Netlist DAGs are shallow and wide: they consist of a large number of short tasks.

3.3 Parallel Full-Cycle Simulation

Multicore parallelism in general-purpose, shared-memory processors is the most common form of parallelism in today's computing platforms. Parallel threads on a shared-memory machine must *synchronize* if they wish to exchange data without race conditions. However, synchronization is costly as it implies communication through the last-level cache (LLC), and so should be kept to a minimum. We do so by performing synchronization at the *coarsest* granularity needed for correct RTL simulation: between clock edges.

We separate each RTL simulation cycle into two distinct phases: *computation* and *communication*. We start by partitioning at compile-time the netlist DAG into multiple independent graphs by creating a DAG per sink node (next register value or memory). The computation in each graph consumes multiple current register values and produces exactly one value in a next register. The DAGs are independent and so can be evaluated in parallel during the computation phase. The computed values are communicated to the DAGs that will consume them in the communication phase. A new simulation cycle starts once communication concludes.

This model is inspired by the *bulk-synchronous parallel* (BSP) execution model [106] shown in Figure 3.2. Synchronization occurs only *twice* per cycle with BSP—once after the computation phase and another after the communication phase.

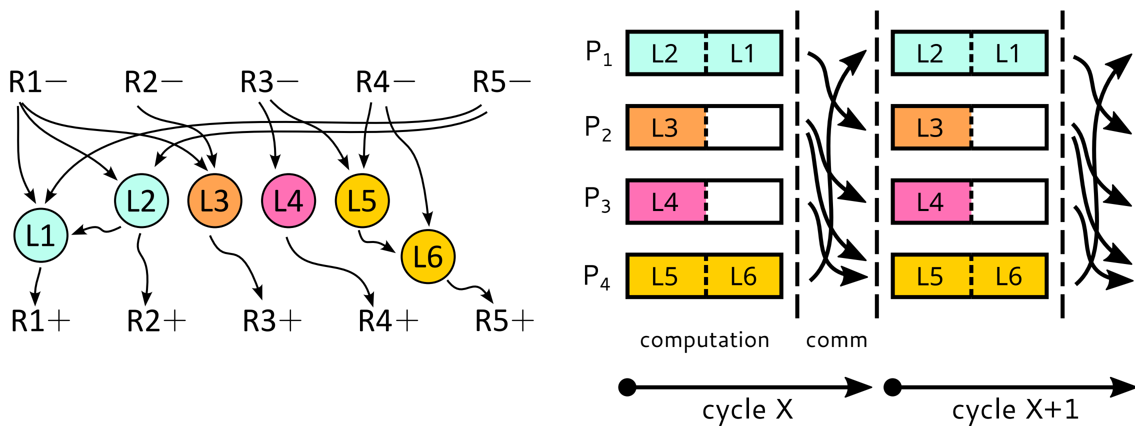


Figure 3.2 – Bulk-synchronous parallel RTL simulation. Each simulation cycle is split into a computation and communication phase. Communication is not all-to-all: producers perform blind writes to only the consumers who need the updated register values.

3.4 Bounding Parallel Simulation on General-Purpose CPUs

Given the abundance of task-based parallelism in RTL simulation, it seems reasonable to expect that the parallel simulation of a circuit exhibits good performance on a multicore machine.

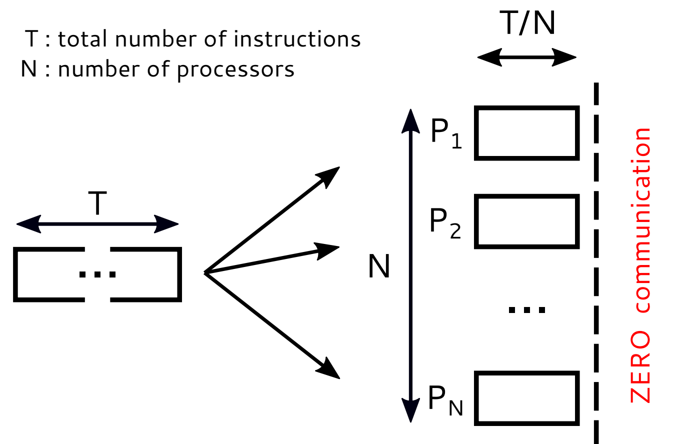
We use a simple model of a simulator to find the relationship between simulation speed, design size, and computation granularity. In practice, simulator speed depends on the target RTL design and details of the simulator's partitioning, optimization, and runtime. A fully accurate model is unnecessary if a simplified model offers an upper bound on any system, which we achieve with three simplifications:

- We ignore the data transfer among cores and focus exclusively on the synchronization *necessary* to coordinate data movement. BSP requires two synchronization points (barriers) per RTL cycle: one at the end of computation and another at the end of communication. These are the minimum synchronization operations needed to simulate an RTL cycle correctly.
- We assume computation can be perfectly parallelized among threads so that there is no straggler.
- As in full-cycle simulation, we assume the number of machine instructions required to simulate one RTL cycle is independent of a design's state.

Figure 3.3 shows our experimental setup: We study parallel simulation performance scalability with a *strong scaling* experiment that increases the number of threads while keeping the total work constant. We explore two variants of the same simulation model, which we present next.

3.4.1 First Model

The bottom part of Figure 3.3 provides a pseudocode description of the first model. The model is ideal as we equally divide a total number of instructions T among N threads. The inner




```
// T is the total number of instructions per cycle.
// N is the number of parallel threads.
// B is an arrive-await barrier.
// C is the number of RTL cycles to simulate.
for N parallel threads {
   localInstr = (T / N) / #instrInWhileLoop
   for C iterations {
      // mock computation
      while (localInstr > 0) {
         localInstr -= 1
         // unoptimizable sequence of independent instructions
         a ^= (a+1)
         b ^= (b+1)
         c ^= (c+1)
         d ^= (d+1)
      }
      B.wait()
      // mock zero-cost communication
      B.wait()
   }
}
```

Figure 3.3 – BSP strong scaling experiment on multicore shared-memory processors. We model how parallel simulation performance scales as a function of a circuit's size and the number of cores used. T denotes the number of instructions needed to simulate one RTL cycle, and N denotes the number of worker cores used. The bottom part gives a more detailed pseudocode description of the experimental setup.

`while` loop executes a sequence of instructions to approximate the simulator's computation of an RTL cycle. The model contains two barriers at the end of this computation to synchronize the communication of newly computed values. These barriers execute when the model runs and contribute to its runtime cost.

We implement the model with care to maximize performance:

- We use multiple unoptimizable, independent instructions in the `while` loop to provide sufficient ILP for a modern processor and avoid core stalls.
- The barriers are implemented as atomic variables. We reduce false sharing by aligning each atomic variable to a 64-byte boundary so they are placed on separate cache lines.
- We pin threads to specific cores to minimize cache migration and interference. We do not use hyper-threads.

Figure 3.4 reports the results of the strong-scaling experiment. We measure the simulation *rate* (top half) and *speedup* (bottom half) compared to serial execution. We target two classes of x86 processors: (1) a high clock frequency desktop processor (i7-9700K), and (2) a high core count server processor (EPYC 7V73X). Details on these processors are available in Table 6.1.

The *dashed* curves in Figure 3.4 represent performance of this first model. We defer a discussion of the results to Section 3.4.3 after we present the second model.

3.4.2 Second Model

The first model does not fully capture the behavior of a simulator since the `while` loop has a small instruction footprint and easily fits in an I-cache. RTL models are typically much larger and incur I-cache misses, in particular as there is little opportunity for reuse in the body of an RTL design. The fraction of a model that runs on a processor depends on the number of threads; hence, the I-cache performance depends on parallelism. We therefore fully unroll the `while` loop to capture this effect, which we report with *solid* curves in Figure 3.4.

The differences between the dashed and solid curves show that simulation speed decreases significantly because of cache pressure.

3.4.3 Discussion

Looking in detail, Figure 3.4 identifies three regions of parallel operation:

- Small circuits (top graphs) contain at most a few thousand instructions to simulate one RTL cycle. Each clock cycle is a small computation and serial simulation can reach rates of a few MHz. Parallel simulation introduces synchronization every 100–1000 instructions (i.e., very fine-grained parallelism) and its cost causes a steep drop in performance between 1 and 2 threads.
- Medium-sized circuits (center graphs) contain between a few thousand to a few hundred thousand instructions per RTL cycle. In this region, synchronization occurs every 2,000–20,000 instructions, so additional threads usually improve performance. However, the performance benefits are limited as, eventually, the synchronization costs outweigh the benefits of splitting the computation further and performance decreases. This region emphasizes the importance of serial performance; the server processor (EPYC) lags behind the desktop processor (i7), even with its many cores and large caches.

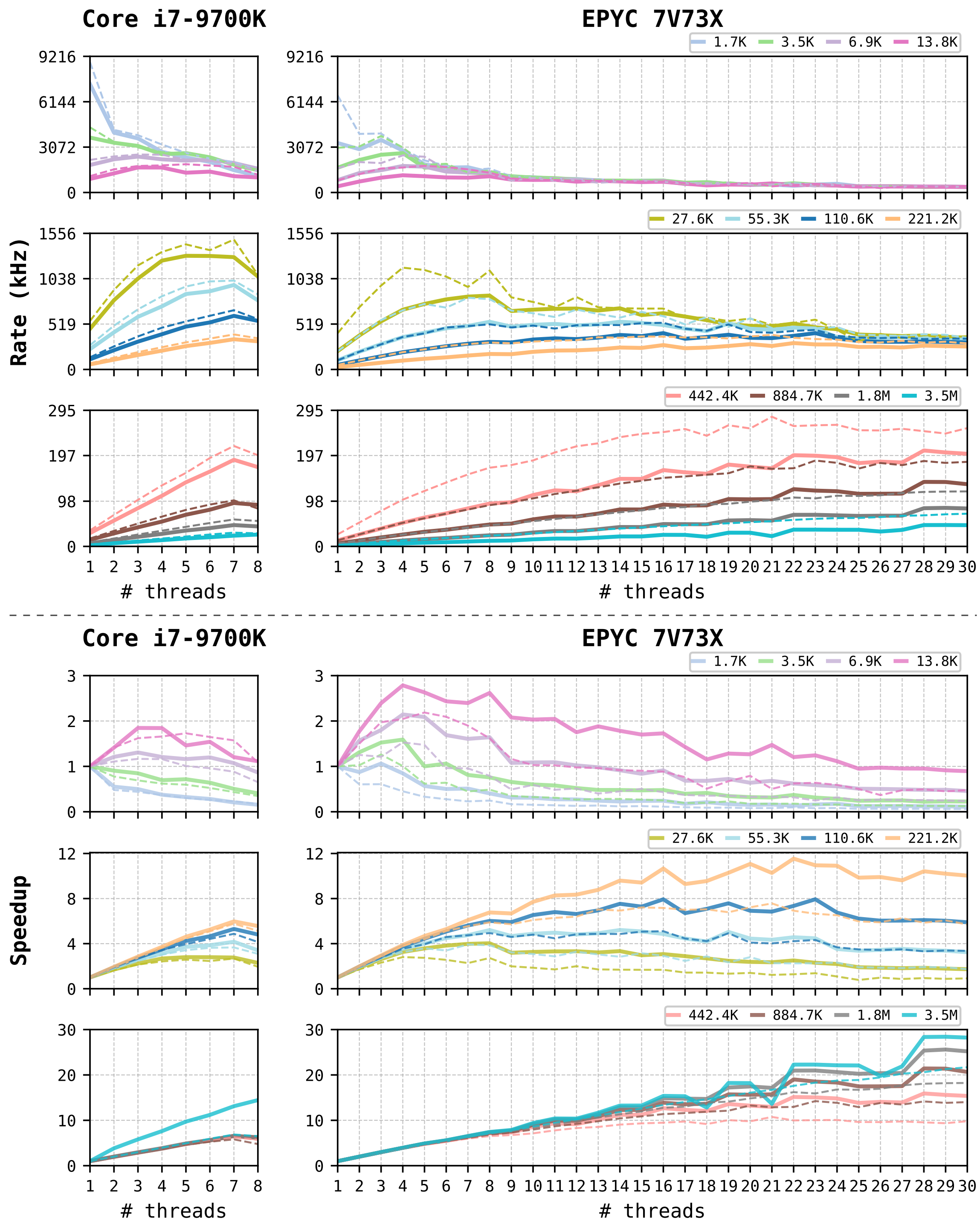


Figure 3.4 – Idealized simulator performance. We report simulation rates (top half) and simulation speedup against serial execution (bottom half). We measure performance on a desktop processor (i7-9700K) and a server processor (EPYC 7V73X). Dashed lines model only synchronization cost (first model). Solid lines also include I-cache pressure (second model). Each curve is labeled by the number of instructions in a simulation step.

- Large circuits (bottom graphs) contain up to a few million instructions per RTL cycle. With large simulation bodies, parallel execution is beneficial since synchronization is infrequent.

However, the overall simulation rate is low because each cycle is costly. Many cores are needed to push the simulation rate into the 100 kHz range, and the simulation benefits from servers' higher core counts.

The figures display numerous inflection points where simulation performance decreases with increasing resources. These inflection points are particularly prominent in fine- and medium-grain simulation (top and center graphs). They occur because additional processors reduce the work-to-synchronization ratio and increase the cost of a barrier. Larger designs (bottom graphs) offer increased opportunities for speedup. The second model's speedups are better since its numerator (serial execution) suffers more from I-cache misses than the first model's smaller kernels. One data point (i7, 3.5M) shows that relieving cache pressure by increasing the number of threads can produce *super-linear* speedup.

Note that the largest design point in our model contains over 3 million instructions in the loop body. Synchronization among even 30 cores results in synchronization every ≈ 300k instructions, which is more than enough work to keep a thread busy. Nevertheless, we observe a flattening of the simulation rate and speedup, which highlights the cost of synchronization—even when infrequent.

We analytically model the parallel simulation rate to gain insight into the limits of parallelism on shared-memory machines. We make two assumptions to simplify the model:

- We assume an ideal processor with a CPI[1] of 1, which means the processor retires instructions at the same rate as its clock frequency F;
- Since there is no global synchronization primitive in general-purpose processors, we assume threads must synchronize iteratively. This means the cost of synchronization for N threads is *at least* $(N-1) \times S$, where S is the cost of a single synchronization through the LLC.

Together these assumptions produce the following relation for the parallel simulation rate r:

$$\overbrace{r(T,N,S,F)}^{\text{sim. rate}} = \frac{\overbrace{F}^{\text{instr. rate}}}{\underbrace{T/N}_{\text{instrs. per core}} + \underbrace{(N-1)\times S}_{\text{barrier cost}}}$$

We find the model's critical point with respect to increased parallelism by differentiating over N and solving for 0:

$$\frac{\partial r}{\partial N} = 0 \iff N = \sqrt{T/S}$$

Parallel simulation on a shared-memory machine will therefore always undergo performance degradation past $N = \sqrt{T/S}$ cores. This result suggests that general-purpose machines can exploit

[1] Clocks Per Instruction

parallelism only through *weak scaling*, i.e., the problem size must increase if one wants to use more cores without experiencing performance degradation.

If we truly want to speed up RTL simulation through parallelism, we need an architecture that can achieve *strong scaling*, i.e., an architecture that can improve performance by using additional cores *without* having to increase the problem size.

3.5 Are Other Architectures Suitable?

Given that RTL circuits exhibit extensive parallelism, it seems natural to think that another one of today's massively-parallel processors could be a suitable platform for simulation.

GPUs are dedicated accelerators with massively parallel architectures that are backed by high-bandwidth memory systems. GPU programs are kernels that describe how individual threads operate on large datasets using explicitly parallel programming models (OpenCL, CUDA, etc.). A thread represents an independent computation that operates on a subset of the problem's dataset. GPUs can support the parallel execution of *thousands* of threads in batches of 32.

GPUs originally supported global synchronization among threads only through kernel invocations by the host processor. Kernel invocations require communication over the PCIe bus, an operation that takes on the order of 1 µs to complete. This call would therefore limit the GPU to a maximum RTL simulation rate of $\approx 1\,\text{MHz}$. However, modern GPUs support *cooperative groups* [43] that allow synchronizing threads within and across GPUs without needing to involve the host processor. GPUs therefore seem like viable candidates for RTL simulation.

Feeding thousands of threads with independent instruction streams at multi-GHz frequencies requires massive instruction bandwidth that greatly exceeds the capacity of memory systems. To make the instruction bandwidth requirements tractable, GPUs *share* instructions among threads that execute the same kernel: GPUs have *data-parallel* functional units where groups of 32 threads execute in lock step using the same code, i.e., GPUs are single instruction multiple thread (SIMT) machines. RTL simulation, however, is not a data-parallel workload. Each tiny task can have a different size and, more importantly, require entirely different operations to represent. It is possible to launch independent instruction streams on a GPU using streams [42], but the lack of data parallelism in RTL simulation means using only a single vector lane in each stream, which results in a resource utilization of only $1/32 = 3.125\%$!

In essence, while RTL simulation exhibits abundant parallelism, it does not have much regularity. GPUs are perfectly suitable for *stimulus-level* parallel RTL simulation: simulating a given RTL circuit with a massive number of input vectors [63], which is very useful for running test suites in batch. However, GPUs are not a good fit for accelerating the execution time of a single simulation, which is the most common workflow while debugging a design or during design space exploration.

3.6 The Manticore Architecture

We now present *Manticore*, our proposed architecture for scalable, parallel RTL simulation. RTL simulation requires an architecture that can support multiple independent instruction streams that act on independent datasets—a MIMD machine. Manticore is a MIMD machine that exploits parallelism in such a way to achieve *strong scaling*.

3.6.1 Key Insight

RTL simulation is a workload that requires frequent synchronization among many cores. Manticore's key insight is to make the cost of synchronization *constant* so it is *independent of parallelism*. We model the effect of this design choice on the parallel simulation rate as follows:

$$r(T,N,S,F) = \frac{F}{T/N + \underbrace{(N-1)\times S}_{C}} \quad \Rightarrow \quad \frac{F}{T/N + C}$$

This function does not have a critical point with respect to N; it *monotonically increases* with respect to the parallelism factor N and the clock frequency F. An architecture with constant synchronization cost therefore achieves strong scaling.

However, recall that our model of parallel simulation rate assumed we ignore the cost of data transfers among cores. This assumption is unrealistic: real workloads will always have shared state that must be communicated among cores at runtime. The cost of this communication necessarily affects the possible speedup.

Adding the total cost of communication among cores, D, to our model of parallel simulation rate produces the following relation:

$$r(T,N,F) = \frac{F}{T/N + C + D}$$

Notice that if D is independent of the parallelism factor N (i.e., it can be bounded by a constant), then the model of parallel simulation rate retains its strong scaling property as $r(T,N,F)$ still increases monotonically with respect to the parallelism factor N.

The following sections provide a high-level overview of Manticore's architecture and how it accomplishes this result.

3.6.2 Architecture

Figure 3.5 outlines Manticore's manycore architecture. It consists of a grid of simple cores that communicate over a network-on-chip (NoC).

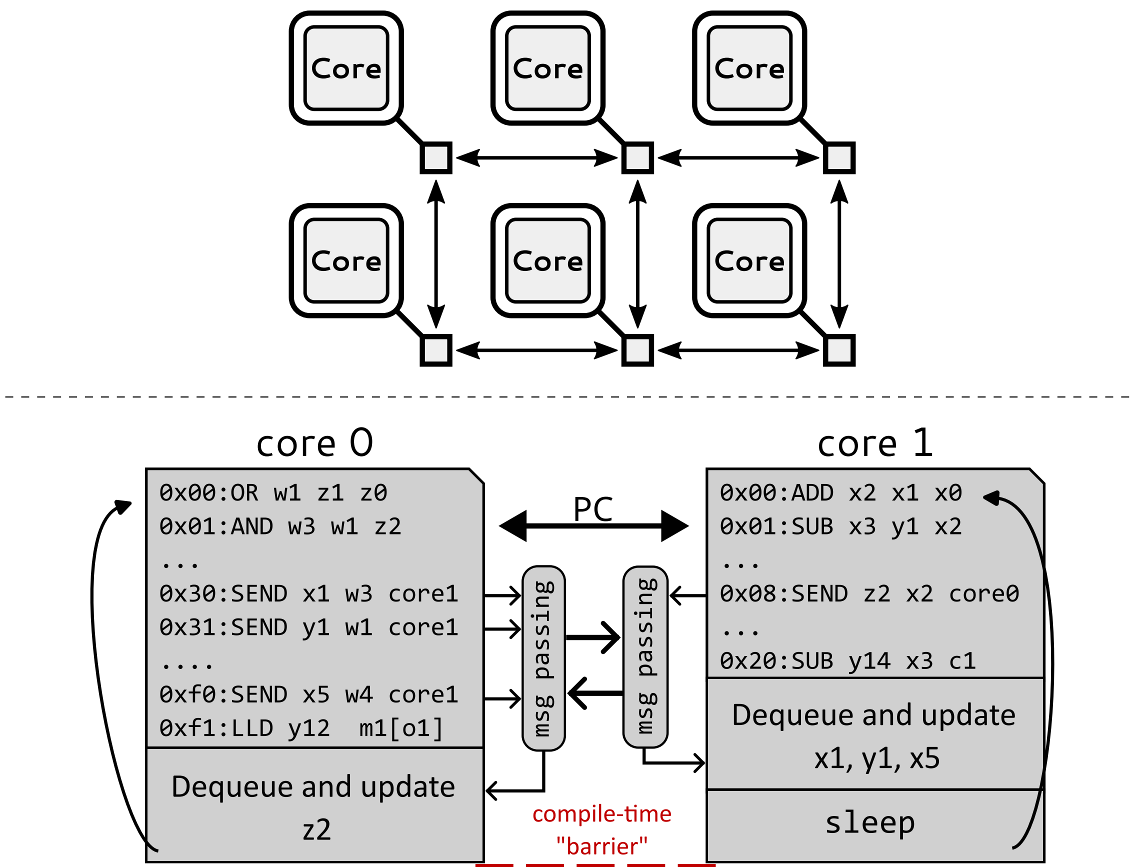


Figure 3.5 – Manticore high-level architecture. Manticore is a manycore MIMD machine: it consists of a grid of cores that are interconnected with a simple network-on-chip. The cores and the NoC execute in strict lock-step and there is no hardware-enforced consistency: an omniscient compiler is entirely responsible for scheduling computation and communication to ensure correct execution on Manticore. The compiler "synchronizes" cores by inserting explicit delay operations (`NOP` instructions in the `sleep` phase) to ensure different processes are in the same phase at all times.

Use compile-time schedule across all cores to remove runtime synchronization entirely

Manticore uses *static BSP* as its parallel programming model. Instead of making *runtime* synchronization more efficient, we use *compile-time* scheduling to remove synchronization traffic entirely. This idea is inspired by compile-time scheduling on VLIW processors. The main difference is that VLIWs apply compile-time scheduling to a single processor to extract more ILP, whereas we apply it to an entire grid of cores to remove the runtime BSP synchronization points between them. The compiler essentially replaces runtime barriers with delay operations (e.g., `sleep` in Figure 3.5 corresponds to a pre-determined number of `NOP` instructions) that ensure all processes start the next BSP phase at the same time.

Beyond scheduling what happens within each core, the compiler also schedules the communication of values among cores. It does so for two reasons: (1) the NoC can avoid buffering, and (2) we can overlap BSP's computation and communication phases to reduce the overall cost of communication.

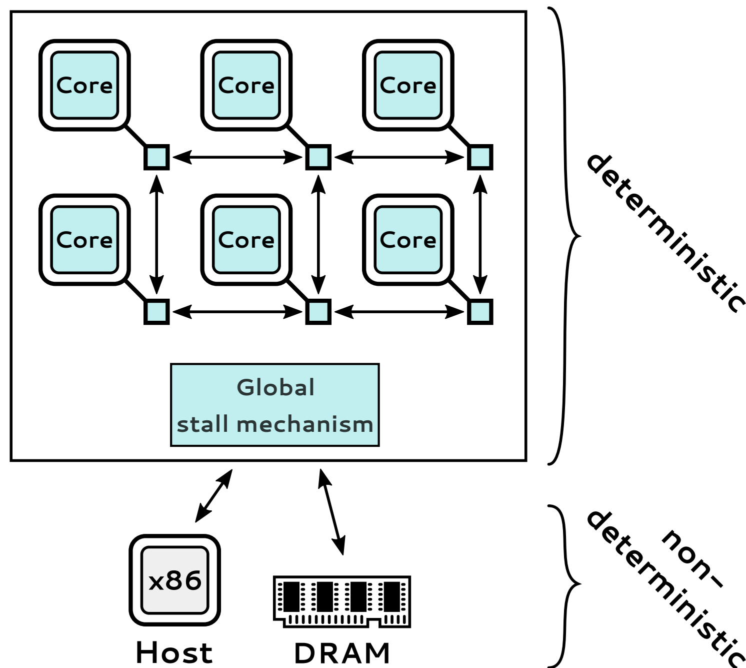


Figure 3.6 – Global stall. Manticore maintains determinism when interacting with the non-deterministic external world by using a global stalling mechanism. It temporarily halts all cores and the NoC while a non-deterministic operation is underway, and resumes their operation once a non-deterministic operation completes.

Compile-time synchronization requires a co-designed compiler and architecture to permit scheduling. In particular, the hardware must expose a *deterministic* interface so that a compiler can accurately track fine-grained computation and communication.

Map circuit state to distributed on-chip SRAM to enforce determinism

Determinism is incompatible with dynamic stalling, which would occur due to contention for a shared instruction memory and caches. We solve this problem by statically partitioning instructions across cores and storing them in *fixed-latency on-chip memories*. Using on-chip memories for instructions eliminates frontend stalls entirely, but requires each program partition to fit in a core's instruction memory.

RTL simulation is a single, long basic block of computation without function calls. We propose using a large, multi-thousand-entry register file in each core to hold its working set so that we can avoid loading and storing values from a data memory. ASICs and FPGAs contain MiBs worth of SRAM, which is enough to hold hundreds of thousands of circuit registers.

Similarly, we use an on-chip scratchpad memory to hold the small (few KiB) RTL memories commonly used in circuits (e.g., FIFOs, etc.).

Global stall when going off-chip to maintain determinism

While ASICs and FPGAs today have a few MiB of on-chip memory, it is not enough to hold very large memory models (e.g., DRAMs, etc.). Large RTL memories therefore cannot be mapped to on-chip memories and we must fall back to off-chip DRAM for capacity. However, DRAM has unpredictable access latencies and is incompatible with compile-time scheduling.

We use a global stalling mechanism to ensure all cores and the NoC remain in lock-step when off-chip access is necessary (see Figure 3.6). The mechanism used for global stalling depends largely on Manticore's implementation, so we defer its presentation to Chapter 4.

Simplify core design to increase core count and clock frequency

The RTL simulation rate is proportional to Manticore's core count and clock frequency, so we want to increase these quantities as much as possible. Following the VLIW philosophy, we simplify cores by removing interlocks and dynamic scheduling circuitry and delegate their functionality to the compiler, which resolves dependencies and schedules instructions at compile time for correct execution. This allows us to design simple, feed-forward, high clock frequency processor pipelines. RTL simulation has extensive ILP, so the compiler can likely fill empty pipeline slots with work. Simple pipelines also result in smaller cores, which allows us to fit more cores on a chip.

3.6.3 Placing Manticore in the System Hierarchy

Manticore is designed as an offload engine for a host processor. Offload engines can either be located within a host processor in the form of a co-processor, or as a dedicated accelerator connected to a system's I/O bus.

A co-processor is a tightly coupled *synchronous* extension to a host processor: it shares resources with the main processor, and data movement between the processors is directly controlled with instructions. However, integrating a co-processor into a processor requires careful design considerations to ensure that they can work seamlessly together. This may involve significant modifications to the processor architecture, which can be time-consuming and expensive. The design space of a co-processor is also limited by its interface: we cannot implement architectural ideas that cannot fit within the framework established by ISA extensions.

A dedicated hardware accelerator is a compute device that is independent from the main processor and sits on an I/O bus—typically the PCIe bus on modern computers. It runs *asynchronously* from the host and is not coherent with host memory; the host is responsible for coordinating its operation using a software *API* and a driver. Since accelerators on an I/O bus are independent from the main processor, designers have increased flexibility when it comes to implementing them.

For these reasons we choose to implement Manticore as an I/O device.

```
int main() {
  // Create design
  top = ...

  while (!finished) {
    // Drive inputs
    top->data = ...;
    top->clk = !top->clk;

    // Evaluate design body
    top->eval();

    // Retrieve outputs
    ... = top->out;
  }

  // Cleanup
  ...
}
```

Listing 3.1 – Offloading only the state update computation is expensive. The round trip time from a host processor to an accelerator that sits on the PCIe bus is on the order of 1 µs, which caps its rate of invocation to 1 MHz. The accelerator needs to generate inputs and check outputs itself to reach higher invocation rates, i.e., the entire loop *body* must run in the accelerator, not just the state evaluation alone (highlighted call to `eval()`).

3.6.4 Interacting With Manticore

Software's sequential execution model has the property that *all* inputs needed to call a function are available at call time. A prerequisite to offloading a computational task to an accelerator is therefore to copy the task's inputs to the device's memory. Only then can software call an API to start the accelerator's operation.

RTL circuits differ significantly from software as they receive inputs and produce outputs *incrementally* over multiple clock cycles. Listing 3.1 outlines the high-level operation of an RTL simulation loop: feeding inputs, evaluating the circuit body, and checking outputs. It is natural to think one can simply offload the parallel part of the simulation—the highlighted call to `eval()`—to achieve good acceleration, but this would limit performance. Suppose the main processor can feed inputs and retrieve outputs with zero cost. The processor's call to `eval()` at each cycle would require communication over the PCIe bus, an operation that takes on the order of 1 µs to complete. This call alone limits the accelerator to a maximum invocation rate of ≈ 1 MHz. In summary, invoking an accelerator from the main processor at each simulation cycle is not an option.

If an accelerator is to achieve true speedups over a CPU, it needs to handle input generation on-device independently from the main processor. For this reason we assume that Manticore's input circuits are *closed*, i.e., they expose only their clock inputs which Manticore can then "tick" autonomously. A target circuit can be closed by wrapping it with a *testbench which itself is a*

circuit. The testbench attests of the functionality of the DUT by driving its inputs and triggering assertions if unexpected behavior occurs during simulation. The netlist shown earlier in Figure 3.1 is a closed circuit. Verilog, SystemVerilog, and VHDL all support assertions that can be checked by a simulator at runtime and stop execution on failures. Having an accelerator that can support large memories now becomes a key enabler for acceleration as making a testbench into a circuit requires reading workloads from somewhere. This location can simply be a large Verilog memory that the accelerator treats like any other circuit memory.

3.7 Summary

RTL simulation is a workload that exhibits abundant fine-grained parallelism, which is incompatible with the coarse-grained parallelism for which general-purpose multicore machines were designed. Multicore machines can effectively use multiple cores for RTL simulation only through weak scaling as increasing the problem size amortizes the overheads of frequent synchronization.

This chapter presented Manticore, an architecture for scalable, parallel RTL simulation. Manticore's key insight is to make the cost of runtime synchronization among hundreds of cores *zero*, which then allows parallel RTL simulation to scale proportionally to the amount of parallelism used. Manticore achieves strong scaling as it can reduce simulation time through parallelism without increasing the problem size.

This concludes our overview of parallel RTL simulation and of the Manticore architecture. Chapter 4 presents an implementation of Manticore.

4 Manticore Microarchitecture and FPGA Implementation

This chapter describes an implementation of the Manticore architecture. Manticore is a grid of deterministic cores that communicate through a deterministic network-on-chip (NoC). The cores collectively evaluate the body of an RTL circuit, then exchange data before starting the next simulation cycle. Chapter 3 showed that Manticore's simulation rate (ignoring communication) is proportional to its core count and to its clock frequency. Our goal is therefore to implement the largest possible grid of cores that can run at the highest possible clock frequency.

We present an FPGA-based implementation of Manticore on a large Xilinx UltraScale+ device (the Alveo U200 datacenter FPGA card). Closing timing at high clock frequencies is challenging, in particular for processor designs because of their control circuitry. However, Manticore's hardware is co-designed with its compiler, which enables us to simplify the cores to increase their clock frequency. Our highest performing Manticore design is a 15×15, 225-core grid that runs at 475 MHz.

We obtained a dense and high-frequency implementation by using low-level FPGA primitives, and so our Manticore prototype's microarchitecture is heavily influenced by what is possible to implement on our FPGA. Section 4.9 discusses alternative FPGA microarchitectures we considered, and Chapter 9 (Future Work) describes possible directions for an ASIC implementation.

4.1 Overview

Figure 4.1 depicts an example, simplified 6-core Manticore grid on the U200 card. Manticore operates as a PCIe-attached accelerator for a *host* (e.g., an x86 processor), which interacts with Manticore through a set of control and status registers (CSRs). The host loads programs on Manticore, handles exceptions or termination, and has full access to Manticore's DRAM. The U200 is partitioned in two areas: the *shell* and the *user design*. The shell is automatically loaded at power-up time. It contains the host PCIe interface logic and auxiliary control structures needed to program the user design portion of the FPGA.

Figure 4.2 shows the floorplan of the U200's large, multi-die FPGA. The FPGA is composed of three super logic regions (SLRs), each containing a grid of clock regions. The shell is provided

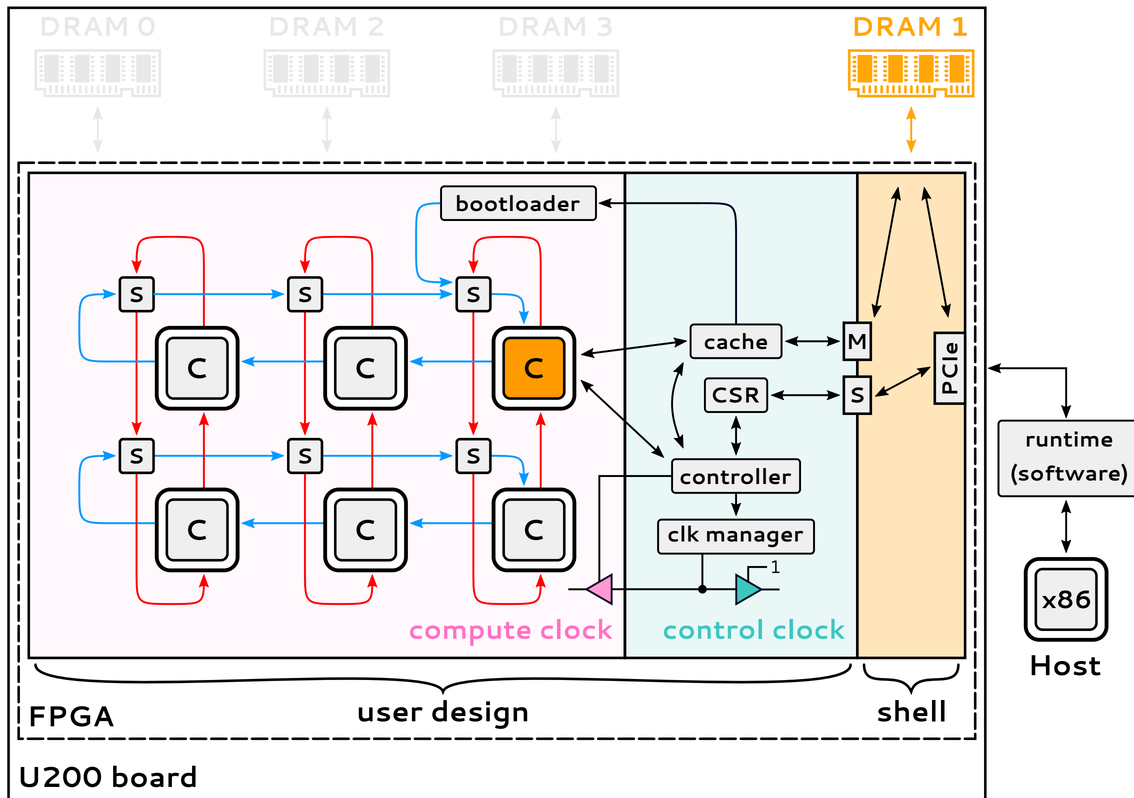


Figure 4.1 – Manticore block diagram. The U200 is a PCIe-attached FPGA accelerator card that is separated into a shell and a user design. Manticore resides in the user design and separates control structures from computational ones. The control structures—clock manager, controller, CSR registers, and cache—reside in the *control* clock domain, while the computational structures—cores (C) and NoC switches (s)—reside in the *compute* clock domain.
One of the cores is *privileged* (orange) and is connected to a cache that has access to off-chip DRAM. The privileged core interfaces with a controller that can pause the compute clock when a non-deterministic operation occurs (e.g., cache miss, exception, etc.) to maintain determinism.

by the FPGA vendor and is *immovable*, so Manticore's implementation must work around it. The U200 has four 16 GiB DRAM banks: one is contained within the shell, whereas the other three are contained within the user design region. Interfacing with a DRAM bank requires instantiating a memory controller, which takes up significant area (highlighted cells within the shell in Figure 4.2). We choose to use only the DRAM bank located within the shell to maximize available space in the user design region for cores.

4.2 FPGA Primitives

Before further discussing the implementation, we provide background information on the FPGA resources that we exploit to achieve a high-performance and dense implementation.

Xilinx FPGAs contain a grid of clock regions. Each clock region contains local and global clock buffers that can be used for either local or global glitch-free clock gating. A subset of the

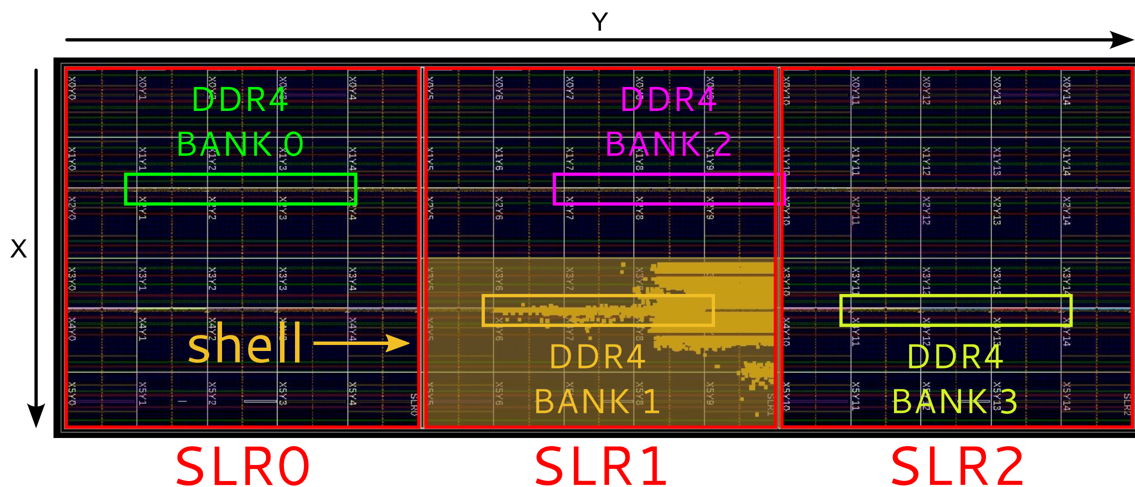


Figure 4.2 – U200 FPGA. The U200 contains a large FPGA composed of three vertically-aligned super logic regions (SLRs). We rotate the figure for space constraints. Each SLR contains a grid of clock regions (gray rectangles). Clock regions with the same X coordinate have the same width in each SLR. The FPGA is separated into a *shell* and a *user design* region. We mark the shell in light orange. The rest of the FPGA is the user design. The device has four DRAM banks. We must instantiate a soft memory controller for every DRAM bank that is used, which consumes a large area (dark orange cells at the corner of the shell). We use only DRAM bank 1 as it is located within the shell and its memory controller's area does not affect the number of cores we can fit on the device.

clock regions contain clock generation primitives that can be used to scale an input clock frequency [100].

Manticore heavily uses on-chip SRAM memories. Xilinx FPGAs contain three types: BRAMs, URAMs, and LUTRAMs. BRAMs and URAMs are dual-port memories with one read and one write port, and with a configurable read latency of 1–2 cycles. BRAMs are 4.5 KiB in size and can be configured using a wide range of data and address widths. For instance, a BRAM can natively implement a 4096×9 (i.e., 12-bit address and 9-bit data) or 2048×18 memory with 1–2 cycles of pipelined read-access latency. URAMs offer 8× more capacity (i.e., 36 KiB) than BRAMs, but support only a 4096×72 configuration.

Lookup tables (LUT) are small, single-bit-wide memories that implement arbitrary logic functions. Some are configurable only at FPGA-programming time, whereas others can be dynamically re-programmed at runtime. The latter are called LUTRAMs and they can be assembled to implement shallow memories [101].

Arithmetic can be implemented either with LUTs (i.e., standard FPGA logic) or with integer digital signal processing (DSP) units. A DSP can be configured to operate as an adder, bitwise logic function, a multiplier, or all three if control pins are toggled appropriately at runtime [105]. If multiple arithmetic operations must be implemented close to each other, a DSP offers a hardened, area-efficient alternative to using multiple parallel LUTs. A DSP's clock frequency can be tuned by configuring its internal pipelining at compile-time.

4.3 Core Design

We now present the design of our high-frequency processor, which is economical in resources and can be replicated hundreds of times on a large FPGA.

4.3.1 Datapath Width

We use a uniform datapath width for the cores and the NoC to simplify physical design and the compiler, so we first determine this datapath width as it influences the rest of the design.

A core's datapath width depends on that of its register file. A small register file can be implemented using flip-flops, which allows supporting arbitrary data widths. However, each core in the Manticore architecture uses a large, multi-thousand-entry register file to hold the circuit registers in its program partition. It is impossible to implement such a register file on an FPGA using flip-flops as there are not enough flip-flops on the device, so we use an on-chip memory instead.

All RTL circuits contain wires of varying bitwidths. In general, most of these wires are narrow (less than 10 bits) and are used for control structures (e.g., multiplexers, decoders, etc.). Each circuit also has another set of wider wires that correspond to the "native" width of the application. For example a circuit that does IEEE 754 floating-point operations will have a large number of 32-bit wires, whereas a circuit that does AES 256 may have 256-bit wires, etc. Using a wide register to implement a narrow wire leads to internal fragmentation in the register file, and consequently to poor register utilization. Conversely, using a narrow register to implement a wide wire increases the program size as more instructions are needed to emulate the wide operation.

URAMs have a fixed 72-bit interface and are wasteful to use for the abundant narrow wires in RTL circuits, so we use BRAMs as they support multiple width configurations. We match Manticore's ALU to the native 16-bit width of DSP units in UltraScale+ FPGAs to reduce register fragmentation for narrow wires and to support a high clock frequency. We then configure BRAMs to be 18 bits wide—the closest supportable width. This results in a 2048-entry register file.

4.3.2 Allocating Resources to Manticore's Building Blocks

Manticore's speed comes from parallelism, so it is important to quantify how many cores we can fit on the U200. If this number is too small, we cannot expect to get any speedup compared to a high-end multi-GHz desktop CPU.

A core is a simple load-store register architecture, so its functional units are a register file, an ALU, an instruction memory, and a data memory (scratchpad). Table 4.1 details the minimum FPGA resources needed to implement these functional units.

We use a single DSP for the ALU to keep a small footprint and support a high clock frequency implementation.

Core FU	FPGA resource		
	URAM	**BRAM**	**DSP**
Register file	0	2/4	0
ALU	0	0	1
Instruction memory	1	0	0
Data memory	1	0	0
Total (1 core)	2	2/4	1
Total available	800	1860	5880
Max cores	**400**	930/465	5880

Table 4.1 – Alveo U200 critical resource analysis. We report the minimum number of FPGA resources needed to implement each functional unit (FU) in a core. Total available resources exclude resources taken up by the FPGA shell. We report the total BRAM usage according to whether a core's register file is implemented using 2 or 4 BRAMs. We omit counting CLB resources per core as CLBs are the primary resource on FPGAs and are abundant. URAMs are the critical resource and limit the number of cores we can fit on the U200 to 400 instances.

A URAM offers 8× as much capacity as a BRAM, so we use a URAM as the instruction memory to support the largest possible program on a single FPGA on-chip memory resource. Manticore's simple 64-bit instructions map directly to the URAMs' 72-bit wide memory interface. This gives us 4096 instructions per core. Similarly, we use a URAM for the data memory as many RTL designs use memories of a few KiB, which can easily be mapped to a single scratchpad.

The simplest register machine is one that can perform binary operations. A processor needs a register file with two read ports and one write port to support stall-free execution in the absence of read-after-write dependencies. URAMs and BRAMs have only one read and one write port, so they cannot implement the register file. We must use two write-replicated units to increase the number of read ports to two. URAMs are our critical resource, so we instead use BRAMs for the register file.

The *minimum* number of resources needed per core is therefore 1 DSP, 2 URAMs, and 2 BRAMs. The FPGA's scarce URAMs limit the maximum number of cores we can accommodate to 400. There are 930 unused BRAMs in the FPGA at this stage, which means we can use up to two additional BRAMs per core without reducing the total number of cores that can fit on the FPGA.

We use the two additional BRAMs to augment the register file into a 4-read, 1 write structure. Doing so allows us to implement a *custom function unit* (CFU) alongside the standard ALU. The CFU is formed of LUTRAMs and supports up to 32 custom functions that can implement any 4-input bitwise combinational function. Manticore's deep pipeline requires a 10-`NOP` gap between dependent instructions. The compiler can fill most of these, but if a pattern is used repetitively throughout a circuit (e.g., in a random number generator), then a custom function can compactly capture its behavior in a single instruction. The CFU reads four inputs from the register file

simultaneously. Section 4.9 discusses alternative architectures we considered to use the extra BRAM capacity.

4.3.3 Instruction Set

Manticore's ISA is simple and contains six types of instructions: (1) standard arithmetic, (2) custom instructions, (3) predication management, (4) local memory access, (5) privileged instructions, and (6) communication. We briefly describe unconventional aspects of the ISA specific to RTL simulation.

Arithmetic instructions include standard two-operand operations (addition, subtraction, bitwise logic, comparison, etc.). They also include a three-operand selector instruction "`MUX rd, sel, rfalse, rtrue`" which copies register `rfalse` to `rd` if register `sel` is 0, and register `rtrue` to `rd` if register `sel` is 1. The standard arithmetic instructions take only register operands, with the exception of (1) the set-immediate instruction "`SET rd, imm`" which updates register `rd` with a 16-bit immediate value `imm`, and (2) the "`SLICE rd, rs, [offset:width]`" instruction which extracts the contiguous `width` bits at position `offset` from register `rs` and writes them into register `rd`. Bit slicing is very common in RTL code, so a dedicated instruction reduces instruction count by avoiding many instances of shift-and-mask instructions needed to emulate its function. A special `ADDCARRY` instruction supports efficient simulation of wide additions by reading and producing an *overflow* bit. Unlike conventional overflow flags, Manticore exposes an overflow bit in all 2048 registers, each of which can be independently read and written. These overflow bits come at no cost as each register in Manticore's register file is natively 18 bits wide. Having a large number of overflow bits gives the compiler more scheduling flexibility.

Each core supports 32 programmable *functions*, which execute chains of bitwise logic operations with up to four inputs in a single cycle. E.g., consider the expression:

```
(a & 0xf) | b | (c & 0x3) | (d ^ 0x1)
```

with a, b, c, and d being operands[1]. A *single* custom instruction replaces these six instructions. Custom functions are programmed into a core during boot.

The "`EXPECT rs1, rs2, eid`" instruction raises an exception `eid` if the values of registers `rs1` and `rs2` differ. Exceptions can invoke services from the host processor (e.g., `$display`). Exceptions, like global memory accesses, stall the execution of all cores and the NoC until they are resolved. Instructions capable of globally stalling the execution are *privileged* and reserved for a single core, which permits an efficient implementation (see Section 4.5).

Each core has a scratchpad memory (up to 128 KiB, limited by the ISA) for local load and store operations. Loads execute unconditionally, but stores are predicated. Global load and store instructions are predicated and privileged, and access large, off-chip memories using 48-bit addresses. From the perspective of a compiler, both global and local memory access have the

[1]Taken from picoRV32, a multi-cycle RISC-V processor.

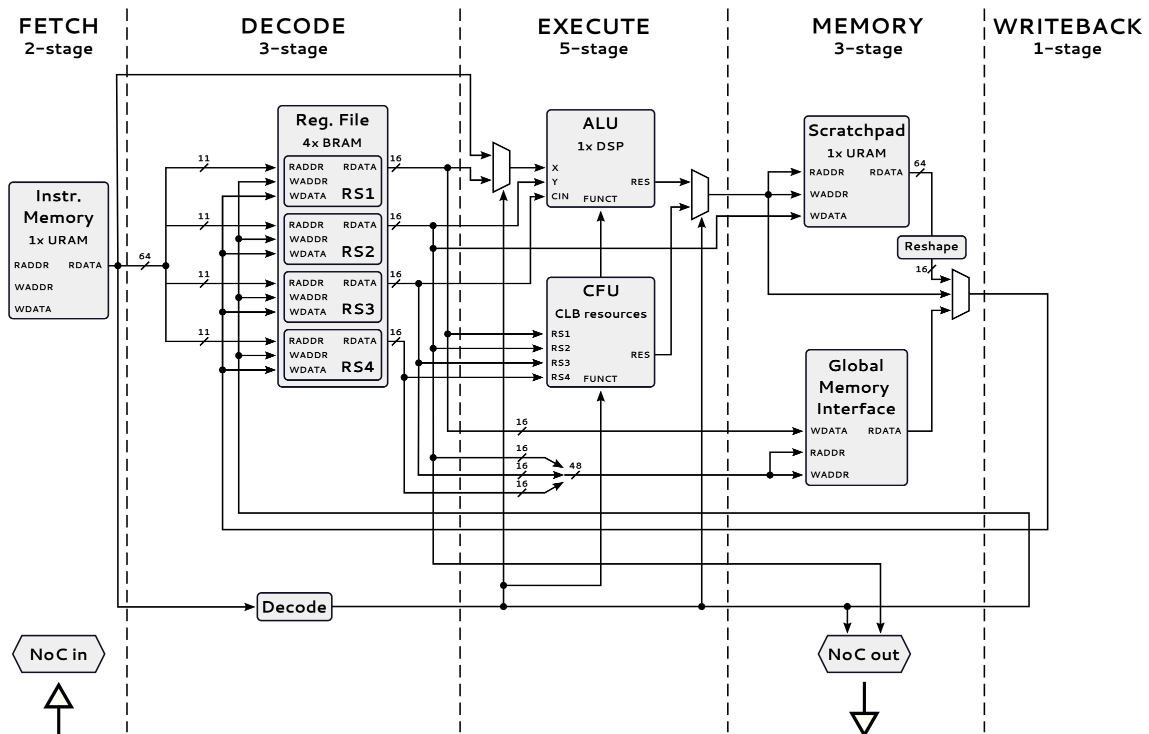


Figure 4.3 – Core pipeline diagram. We show only the datapath and omit all control signals and pipeline registers for legibility. The core has a simple, 14-stage pipeline. The instruction memory and scratchpad are implemented using one URAM each. The register file is implemented using four write-mirrored BRAMs. The ALU is implemented using a DSP, which performs all non-custom arithmetic instructions and computes addresses for local memory accesses. Global memory addresses are 48-bit wide and are concatenated from three registers. The CFU is implemented using standard CLB resources and handles custom instructions. The NoC egress path is indistinguishable from memory accesses and is located at the memory stage. We defer details of the NoC ingress path to Section 4.4.

same predictable latencies since the long off-chip latency is masked by stalling all cores and the NoC until a memory access completes.

The producer of a value initiates communication with a SEND instruction, which is the only way cores communicate. The "SEND `rt, rs, tid`" instruction invoked by a core `sid` requests target core `tid` to update its register `rt` with the value of register `rs` from core `sid`. As Figure 3.5 illustrates, SENDs occur intermixed with computation, but the register updates are delayed until the end of an RTL cycle.

4.3.4 Pipeline Implementation

Figure 4.3 presents a high-level view of the core datapath, which is a simple 14-stage pipeline. The pipeline is simple because we remove expensive bookkeeping logic (e.g., interlocks and scoreboards) and delegate their functionality to the compiler. The pipeline is therefore purely feed-forward and cannot dynamically stall, i.e., the compiler must schedule instructions to ensure correct execution.

The logical pipeline consists of the usual five stages: fetch, decode, execute, memory access, and writeback. Each stage is internally pipelined to achieve a high clock frequency.

Fetch Instructions are fetched over two cycles from a dedicated instruction memory mapped to a 4096×64 URAM. All instructions are encoded in 64-bit words. We sacrifice space and avoid variable-length encoding to keep decoding simple. Most of the instruction space is used to index four 11-bit register identifiers needed to index the core's large register file.

Decode Deep pipelines require a large register file to avoid stalls. Manticore's cores provide a *2048-entry* register file that exposes all registers to the compiler to avoid expensive hardware renaming logic (similar to the MIT Raw machine [108]). The register file is built with BRAMs and is 18 bits wide (the most-significant bit is unused). We use the lower 16 bits to contain register values and the 17th bit as an *overflow bit* for wide addition instructions. Having a large number of overflow bits gives the compiler more scheduling flexibility. The size of the register file requires additional pipelining for reads. Decoding is simple with a set of parallel comparators, but is three stages long as a result of the extra register file pipelining. Custom instructions can read four values from the register file and write a single result. This requires four read ports and one write port, which BRAMs do not natively support. We use four identical BRAMs that are write-mirrored to produce four values simultaneously.

Execute The execute stage consists of two computational units pipelined over five stages. The ALU handles most standard instructions using a hardened FPGA DSP. It is also responsible for computing addresses for accesses to the scratchpad. The custom function unit (CFU) handles custom instructions and is implemented as a small 32×256 memory made of LUTRAMs (see Section 4.3.4). The execute stage contains a global predicate register (not shown in Figure 4.3.4), which is set by a previous PRED instruction and is used for predicated instructions (local stores and all privileged instructions).

Memory Each core has a scratchpad memory for local load and local store operations. The scratchpad is mapped to a URAM, which needs two cycles to access and one cycle to reshape. We reshape a 4096 × 64 URAM into a 16384 × 16 memory by using byte-strobes on the write path and multiplexers on the read path. Local loads execute unconditionally, but local stores are predicated. Global loads and stores are both predicated. Global memory accesses are privileged and access large, off-chip memories using 48-bit addresses formed by concatenating three registers. Only the privileged core can access global memory. Local and global memory accesses are indistinguishable to the compiler as both types of accesses are performed in the same pipeline stage. Global memory instructions can have unpredictable access latencies, but this latency is masked by stalling all cores and the NoC until the access completes (see Section 4.5).

Writeback Finally, the writeback stage propagates the results of the ALU, scratchpad, or global memory loads to the register file.

Note that the core implementation is excessively deep and could be reduced by at least two stages, but layout constraints in our FPGA prevented us from doing so. The main issue is the iMem → RF and dMem → RF distance, which depends on how far URAM columns in the FPGA are located from BRAM ones. Depending on where the placer puts a core, this URAM → BRAM

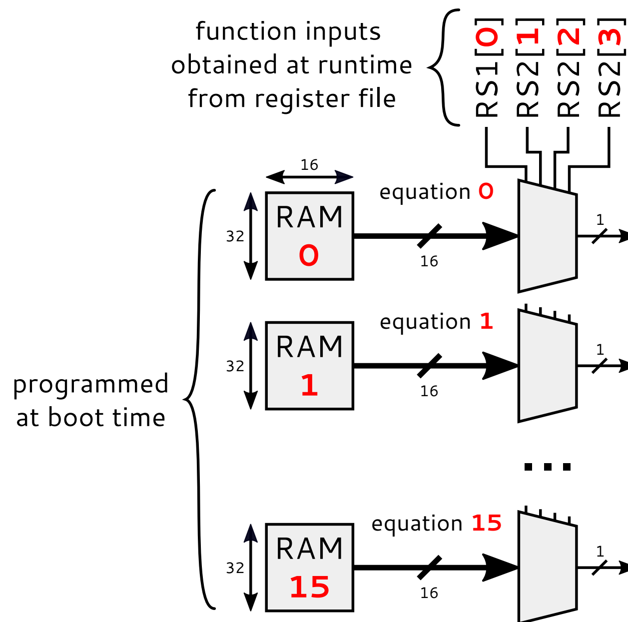


Figure 4.4 – Custom function unit. Each bit of Manticore's 16-bit datapath supports up to 32 4-input custom functions, which each require 16-bit truth tables to represent. We store these truth tables in 16 instances of a compact memory primitive (RAM32M16). Each memory holds truth tables for the ith bit in Manticore's datapath. The functions' inputs are provided at runtime from the register file.

distance can either shorten or lengthen. We therefore had to use additional pipelining after the URAMs to ensure all instances of the cores—no matter where they are placed on the FPGA—can meet our target clock frequency.

Custom Function Unit

The CFU implements arbitrary bitwise-parallel logic functions. A 1-bit, k-input boolean function is canonically defined by the 2^k bits of its truth table. Manticore's datapath is 16 bits wide, so it uses 16 parallel 2^k-bit truth tables to represent a custom function. Manticore can read four inputs from its register file, so $k = 4$ and each truth table needs 16 bits to represent.

Figure 4.4 shows a high-level block diagram of the CFU. The truth tables are loaded into shallow memories at boot time and are indexed by inputs provided by the register file at runtime. We use the largest available 16-bit wide LUTRAM-based memory primitive (RAM32M16) available in UltraScale+ FPGAs to compactly store the truth tables for each of the CFU's output bits.

4.4 Network-on-Chip Design

NoC designs fall in two categories: direct topologies (ring, mesh, torus) and indirect topologies (crossbar, butterfly, clos, fat tree). We used a direct topology for simplicity.

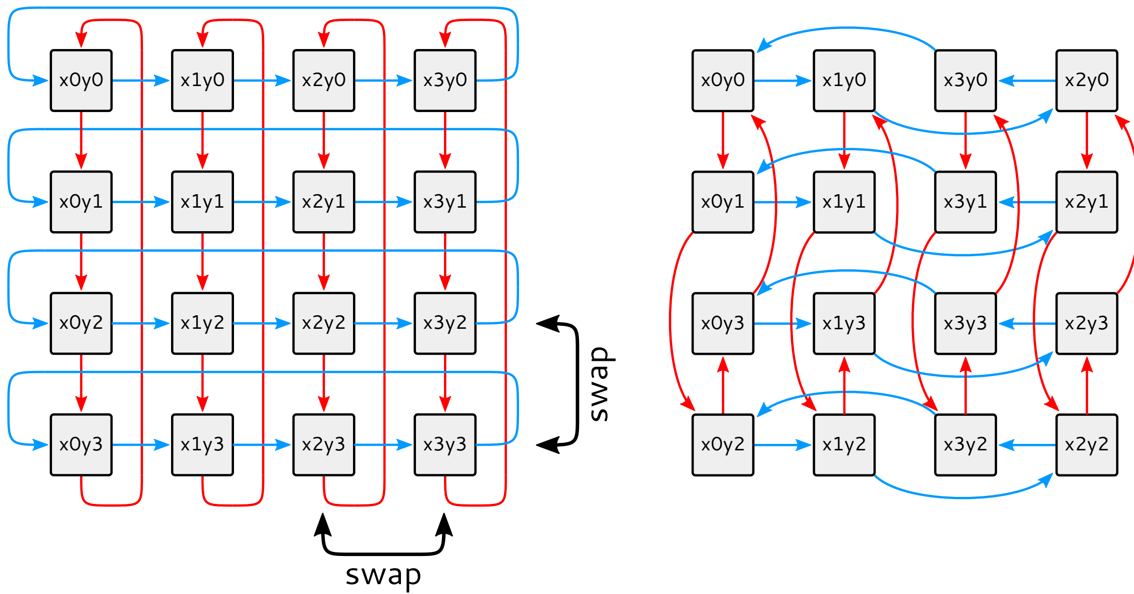


Figure 4.5 – Uni-directional NoC layout. Each rectangle represents a NoC switch (we omit cores for simplicity). We highlight horizontal links in blue and vertical links in red. A uni-directional torus is most easily illustrated using the topology on the left. In practice, the torus is folded on itself to ensure logically-adjacent nodes are placed in a physically-adjacent configuration to achieve a high clock frequency. In this 4×4 configuration, this amounts to swapping the order of the last two columns, and similarly swapping the order of the last two rows.

We ruled out a ring-based design as its 1D structure would result in excessive latency in a multi-hundred-core system. We also ruled out a mesh design: while its 2D structure decreases inter-core communication latency, the nodes at the boundary of the network would have a lower degree than those at the center, which would result in higher demand for the center channels [49]. We therefore used a 2D torus design. We decided to use a *uni-directional* implementation early in the design process as we predicted there would be extensive routing congestion in the FPGA when used near its maximum capacity.

Figure 4.5 illustrates the NoC topology. We typically illustrate the torus structure as shown on the left side of the figure. However, implementing the NoC in this manner causes the wrap-around links to be significantly longer than inner links, which would yield a low clock frequency. Instead we physically fold the torus on itself by interleaving its rows and columns to produce the layout shown on the right side of the figure. Doing so ensures logically-adjacent NoC switches are also physically adjacent, resulting in a high clock frequency implementation.

The NoC uses buffer-less switching and dimension-order routing [53]: its switches are composed entirely of stateless pipeline registers and multiplexers to be area-efficient and achieve a high clock frequency. We chose this design to minimize NoC area and allocate as many resources possible to computation (i.e., core count). All messages are point-to-point and the NoC does not support broadcast: messages are first routed in the X dimension, then in the Y dimension. Dimension-order routing means the switches need not contain a routing table and simply use a

static routing mechanism. Links carry 27 bits of payload, and a few[2] bits to specify the target core address.

The consequence of these design choices is that switches do not queue messages and must immediately route them horizontally, vertically, or to the local core. A switch whose links are busy simply drops the input message. To avoid data loss, the compiler ensures timely delivery by scheduling the `SEND` instructions in each core. Manticore's deterministic execution makes it possible to analyze and predict link utilization at each cycle.

4.5 Global Stalling Mechanism

Manticore's design has remained deterministic until this point, which allows a compiler to schedule computation and communication globally. However, instructions with non-deterministic latency are unavoidable (off-chip DRAM accesses and exception handling), so Manticore needs a global stalling mechanism to maintain determinism when they occur. Stalling involves *immediately* halting all cores and the NoC, processing the non-deterministic operation, and finally releasing the stalled units.

Note that we must stall both the cores and the NoC. Stalling the cores alone is insufficient as the NoC switches are buffer-less and will continue to push messages to their target core. A stalled core is incapable of accepting an incoming message, which would lead to its loss and an erroneous execution.

We originally envisioned a design where all cores are equal and where instructions with non-deterministic behavior can be mapped to any core by the compiler. However, stalling hundreds of cores using a distributed mechanism is challenging, especially if multiple cores execute an instruction with non-deterministic latency at the same cycle. We therefore simplify the design and modify the compiler such that it co-locates all RTL processes that can lead to non-determinism onto one core—the privileged core.

Routing a global enable signal from the privileged core to the rest of the design does not scale to hundreds of cores: since the stall must occur immediately, the enable signal would need to be combinational and span the entire FPGA, which would severely limit the maximum clock frequency of the design.

Instead, we use *clock gating* as the stalling mechanism, which we implement using the FPGA's clock gating primitives (see Figure 4.1). Clock gating provides a single point of control for stalling and scales to logic that spans the entire FPGA as gated clock buffers are located directly on the device's clocking network.

We implement Manticore using two clock domains: all parts that operate in strict lock-step (the cores and the NoC) reside in the *compute* clock domain, while the rest of the logic that deals with non-determinism resides in the *control* clock domain. The logic in the control domain can *halt* or *resume* the compute clock with a global clock buffer. The control clock never halts.

[2] Varies based on the grid size, e.g., 8 bits for a 15×15 grid.

We configure the clock generator and design the control path of the clock gating circuitry carefully to minimize its effect on scalability:

1. The two clocks are sourced from the same clock generator, which we configure to generate two frequency-matched and phase-aligned clocks.
2. There is no logic delay between the controller-driven clock enable signal and the clock buffer.
3. Clock skew between the control and compute clock domains is inevitable given that we cannot use clock crossing logic due to its non-zero latency. To minimize skew, we manually guide Vivado, Xilinx's place-and-route tool, to select a clock region for routing the global clock signal.

The result is that clock gating is nearly independent of the number of cores.

With global clock gating, computation is frozen on a global memory access and resumed once it completes. The same mechanism is used to stall the compute domain when an exception occurs so that exceptions are precise. Control is then transferred to the host machine and computation resumes at the host's command.

Note that the bootloader could interchangeably be in either the compute clock domain or the control clock domain. We ended up placing it in the compute clock domain (see Figure 4.1) as the cores and NoC can intercept messages only when the compute clock is active.

4.6 Off-chip Memory Access and Caching

Each core's local scratchpad can hold RTL memories up to 32 KiB in size. Larger RTL memories do not fit, in which case Manticore falls back to a 16 GiB off-chip DRAM bank on the U200. The compiler knows which RTL memories cannot fit in on-chip SRAM and ensures all RTL processes that utilize such memories are mapped to the privileged core, which is the only core that supports global memory accesses.

DRAM access is costly and its latency is highly variable depending on the access sequence, so we use a cache between the privileged core and the DRAM controller to absorb its latency. We use a simple 128 KiB direct-mapped cache with a write-allocate, write-back policy. The cache is implemented using 4 URAMs[3] and is clocked by the control clock. Each cache line is 256 bits wide.

In our current design the cache *conservatively* halts the clock upon receiving a global memory access request as it does not yet know if the access is a hit or a miss (the privileged core emits global memory accesses at the same place as local memory accesses—in the Memory stage). If the access hits in the cache, then the compute clock is resumed on the next clock cycle. If the access misses in the cache, then the compute clock remains halted until the DRAM access is complete. Therefore, from the compiler's point of view, a global memory access appears to all cores as a fixed latency operation independent of cache and DRAM latency. We could remove

[3]The cache reduces the U200's theoretical maximum core count in Table 4.1 from 400 cores to 398 cores.

this conservative stall by emitting the global memory access one cycle earlier in the privileged core (in the last Execute stage), but we decided to leave it in the Memory stage for simplicity.

The cache is particularly effective at capturing locality exhibited by certain large RTL memories (e.g., FIFOs). We evaluate this later in Section 6.

4.7 Manticore Runtime, Bootloading, and Processor Execution

Each core's instruction memory, register file, data memory, and CFU must be programmed before execution starts. The register file must contain the initial values of RTL wires and various constants needed to emulate them. The data memory must contain the initial contents of RTL RAMs and ROMs. The CFU must contain per-bit logic equations for each of its custom functions. Finally, the instruction memory must contain the actual program to execute. Each of these memories needs dedicated programming circuitry.

In reality, we use the fact that two of these memories are architecturally exposed to remove their dedicated programming circuitry:

- The processor can write its register file with set-immediate instructions.
- The processor can write its data memory with local store instructions.

We therefore need programming circuitry for only the CFU and the instruction memory. We extend the ISA with two additional instructions to program the CFU such that all core internals can be initialized through instructions. The instruction memory itself, however, requires an external programming mechanism. The bootloader (see Figure 4.1) is in charge of this task.

The bootloader works in conjunction with Manticore's runtime, which runs on the host processor. The runtime takes a sequence of binaries generated by the compiler and copies them into FPGA DRAM. The runtime then configures Manticore's control registers with a pointer to the location in FPGA DRAM where the program binaries are stored. It then launches the bootloader which copies the programs into the cores' local instruction memories and starts program execution. While the code executes, the runtime continuously polls the hardware state registers to handle exceptions or terminate execution.

Figure 4.6 shows the NoC ingress path, which is used as part of each core's bootloading process before simulation starts, and for register updates during program execution. We first present the bootloading datapath, then the program execution datapath.

4.7.1 Bootloader Execution

The red datapath in Figure 4.6 describes the core's boot sequence. The bootloader starts with a soft reset that brings all cores to a boot state. The soft reset is not a full system reset as it just changes a few state registers in each core (none of the processor's functional units are reset).

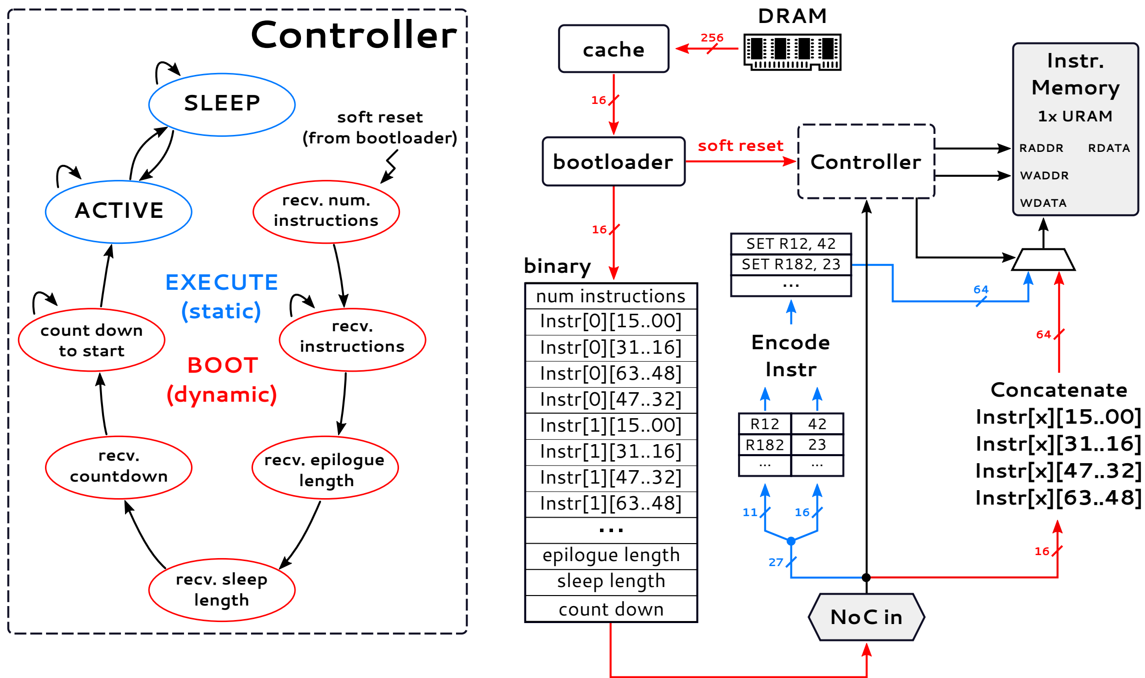


Figure 4.6 – Core NoC ingress datapath. The cache and bootloader are external to the cores. All other components are contained within a core. We highlight the bootloading datapath (red) and program execution datapath (blue). The bootloader streams binaries from DRAM to each core over the NoC before program execution starts. A local controller in each core intercepts bootloader messages, then writes the instruction memory and initializes various core-dependent counters (epilogue length, sleep length, countdown length). During program execution incoming messages are translated to set-immediate instruction on the fly and are queued at the end of the instruction memory. The register updates are then executed by the core as any normal instruction when its program counter reaches them.

Cores are in a “dynamic” state at this stage: they snoop the NoC for instructions and continually push NOPs through their pipelines.

The bootloader then reads the program binary from FPGA DRAM through the cache and injects its contents into the NoC at the privileged core’s NoC switch. Messages then flow to their respective cores. The program binary contains multiple fields, which a controller in each core intercepts and uses to configure the processor.

The controller starts by receiving a counter which determines the total number of instructions that the core will receive. Each instruction is received as four 16-bit chunks, which the controller assembles into a single 64-bit instruction and writes to the instruction memory. Once all instructions are received, the controller then receives three counters.

The first counter is the epilogue length, which is the number of messages the core is expected to receive from other cores during program execution. The second counter is the sleep length and corresponds to the number of clock cycles each core must sleep after the computation phase.

At this stage all cores have the information they need to execute the program. The last step is to start all cores *at the same time* such that they execute the program in strict lock-step. The

bootloader ensures this by sending a custom countdown timer to each core in one sweep. The deterministic execution of the NoC allows these messages to arrive at exactly the right time and the countdown value is accurate. Each core counts down and starts "static" program execution once this counter reaches 0.

4.7.2 Program Execution

While BSP technically separates computation and communication, in practice Manticore overlaps their execution to reduce overall runtime. If a computed *next* register value is finalized, then the source core can send it through the NoC to all its consuming cores for use in the *next simulation cycle*. It is essential that these updated values not be applied to registers at the consuming cores until the end of the computation cycle. Each core therefore needs a queue to hold incoming register updates and delay their application, but we do not want to allocate extra on-chip memory for it as that would reduce the core count.

Each core's instruction memory is a URAM and has one read and one write port. The write port is unused after the bootloading phase in which the instruction memory is programmed by the host through the bootloader. We can therefore repurpose the now-unused URAM write port as a queue to save resources. Inbound messages during program execution are translated on the fly as set-immediate instructions ("`SET rd, imm`" in Figure 4.6, where `rd` and `imm` come from the NoC interface). The instruction is then pushed to the end of the instruction memory, which the core then executes like any other at the end of a simulation cycle.

Each core transitions between two states during program execution: active and sleep (blue states in Figure 4.6).

Cores start the active phase by initializing a total program counter (TPC) to the total number of instructions in their program body plus the epilogue length (which represents the number of messages the core is expected to receive at runtime). If the core's program body is too short, the compiler pads it with `NOP`s until the time of the first message's arrival (which is known at compile time). This ensures incoming messages are written to the instruction memory before the core's program counter reaches them. All subsequent messages from that point are dynamically inserted through register updates from other cores. The processor's controller keeps track of the queue's tail address, which it auto-increments upon receiving a NoC message. Cores execute instructions until the TPC is reached, then they enter the sleep phase.

Upon entering the sleep phase, the cores initialize a timer which they decrement at each cycle. Cores then push `NOP`s down their pipeline until the timer reaches zero, at which point the cores transition back to the active phase.

4.8 Floorplanning

A single instance of the core presented in Section 4.3.4 can be implemented at a 500 MHz target clock frequency on the U200 FPGA using Vivado's automatic place-and-route (P&R) flow. What

Grid	8×8	10×10	12×12	15×15	16×16
Auto	500	485	480	395	180
Guided	–	–	500	475	450

Table 4.2 – Manticore implementation results. We report achieved clock frequency (MHz) on the U200 with automatic and guided floorplanning. We use guided floorplanning for only large Manticore grids as automatic floorplanning is good enough for small configurations.

remains is to scale the design to as many cores as possible on the FPGA, while attempting to keep the clock frequency as close as possible to 500 MHz. This is an exercise in floorplanning: a bad floorplan heavily decreases performance, whereas a good floorplan incorporates more logic in the same total area at higher performance.

Note that it is likely a single core could be clocked at more than 500 MHz given its deep pipeline and simple feed-forward structure, but we explicitly chose not to evaluate higher frequencies. Our past experience with FPGAs taught us that a replicated high-frequency unit spanning an entire device would generally not maintain its clock frequency due to (1) design limitations in the connections between individual units, and (2) routing congestion on a packed device.

Our general philosophy during floorplanning is that it is impractical to manually place individual cells in the design, as making even a single change in the future would be a nightmare. Instead we perform floorplanning by generating Tcl scripts that constrain the design using information about the target FPGA. Manticore is developed in Chisel [9], a DSL for hardware description embedded within the Scala programming language [73]. Manticore's hardware generator is aware of cores' names and those of their internal structures, so we can enumerate cells and emit constraints in Scala as part of Verilog generation. We add a generic Scala floorplan generator class after Chisel's final Verilog emission point, which we subclass to implement various floorplanning strategies.

4.8.1 The Shell's Impact on Clock Frequency

The first row in Table 4.2 shows the achieved clock frequency for Manticore grids that are scaled "as-is" without any floorplanning on the U200's FPGA. Designs up to 64 cores scale effortlessly to 500 MHz. A slight decrease in clock frequency occurs as the design size increases to 144 cores, which can run at 480 MHz. Performance decreases sharply after this point as 15×15 designs drop to 395 MHz, and 16×16 designs drop down to 180 MHz.

This drop in performance is explained by the shell's placement: With fewer than 160 cores (maximum capacity of one SLR, limited by its URAM capacity), Vivado automatically fits Manticore entirely in SLR2, undisturbed by the shell. The achievable clock frequency naturally degrades as the number of cores approaches the SLR's capacity due to increased routing pressure. Past 160 cores, the design is forced to spread around the shell, which makes timing closure difficult.

Vivado cannot automatically find a good floorplan for a rectangular core array in a non-rectangular user design region, so we must guide it to a satisfactory solution. The second row of Table 4.2 shows the achievable clock frequency for Manticore grids following the floorplanning strategy that we present later. Performance is consistently improved, especially in large configurations.

4.8.2 High-level Floorplanning

We must take into account three physical constraints imposed upon us by the U200's design if we want to approach a 500 MHz clock frequency at scale:

1. The FPGA's resource columns are not homogeneously spaced across the device: while all clock regions contain BRAM columns, only a few contain URAM columns.
2. The central SLR in the U200 has fewer resources than its neighbors as the shell takes up half its available space.
3. Each SLR is essentially a small, independent FPGA inside a larger device. Adjacent SLRs are stitched together with silicon interposers, which results in inter-SLR links having higher routing delays than intra-SLR ones.

Given Manticore's regular design, it is natural to believe that a grid-structured floorplan would produce the best clock frequencies. We first show why such a regular floorplan cannot do so on the U200. We then perform a minor design modification that permits an unintuitive, irregular, split floorplan to achieve significantly higher clock frequencies.

Note that we provide a high-level description of the key ideas in each floorplan, but intentionally simplify figures and omit full details that are necessary to implement them in practice as the description would otherwise be unnecessarily long.

Regular Device-wide Grid

Figure 4.7 gives a bird's-eye view of a regular floorplan that attempts to retain as much of Manticore's conceptual 2D grid structure.

We enforce the grid structure by constraining the placement of each core, which we do by constraining the placement of its primary resources; a core's remaining resources will then naturally cluster around them. Manticore's design is limited by the U200's available URAMs and BRAMs (there is an abundance of CLBs and DSPs), so we need constrain the placement of these memories. Mapping an RTL name to a specific FPGA cell over-constrains the design and generally yields inferior results as Vivado then has little flexibility. We instead assign a core's resources to coarse-grained bounding boxes (BBs) and let Vivado handle detailed placement.

We cannot use the U200's existing clock region boundaries as BBs since many clock regions do not contain URAM columns, which would prevent cores from being placed in them. We expand the BBs to ensure enough resources are available for a symmetric row-based core layout. We do so by creating a "left" and "right" BB per row of clock regions in the U200 (top of Figure 4.7). We then iteratively assign cores to each "left" and "right" BB—assigning proportionally fewer cores

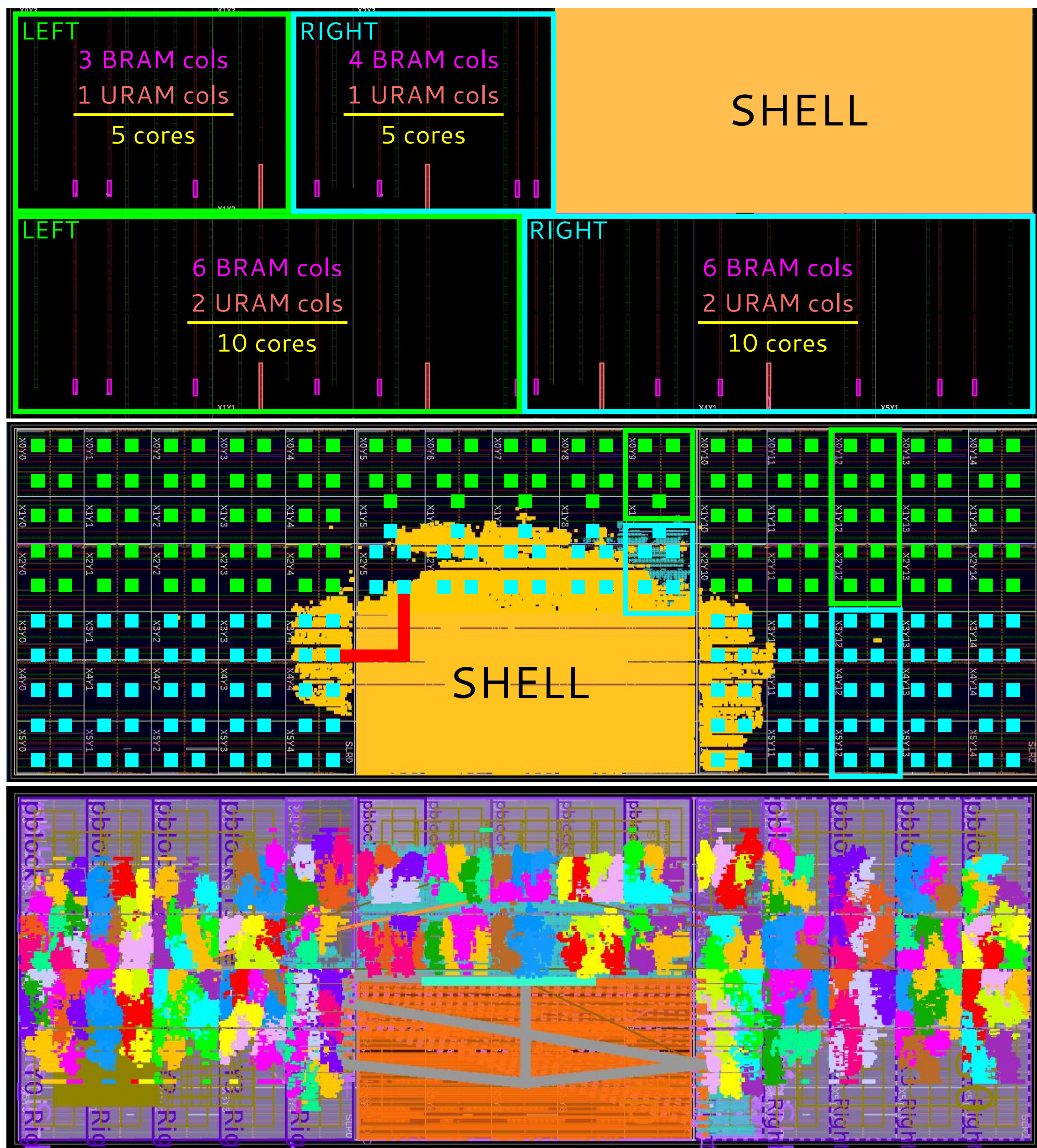


Figure 4.7 – Regular placement of a 25 × 10 Manticore grid. We identify a "left" and "right" bounding box (BB) in each of the U200's clock region rows (top). We assign a set number of cores (green/cyan squares) per BB given its capacity (center). This allows fitting a large 25 × 10 Manticore grid on the U200. The cores naturally cluster around the central columns of the device where URAMs are located (bottom, we color adjacent cores differently to highlight their position). The thick red line in the center figure is typically the critical path of the design.

to the narrow SLR to account for its reduced resources—to obtain a regular 25 × 10 Manticore grid (center of Figure 4.7). The bottom of Figure 4.7 shows the final floorplan, where each color represents a different core.

While the design is regular, we observe that cores cluster around the center columns of the device despite having created BBs that span the full available width in each SLR. This behavior is explained by the position of scarce URAM columns, of which only four are available in the device (top of Figure 4.7). The shell's footprint covers two of the URAM columns in the central SLR, despite not using them. As a result, the narrow SLR is, in particular, quite cramped as it contains (1) cores, (2) NoC switches, (3) auxiliary control structures in the control clock domain (cache, clock manager, etc.), and (4) shell interfacing logic that Vivado auto-generates and which protrudes past the shell's boundaries. The resulting increased routing congestion creates long paths between switches around the shell, further compounded by the increased inter-SLR crossing (thick red line, center of Figure 4.7), limiting the achievable clock frequency.

Obtaining a design that can be clocked near 500 MHz requires at least relaxing routing pressure in the narrow SLR. Alas the Xilinx-provided shell is immovable and designing an alternative shell that does not occupy the center SLR's URAM columns is beyond the scope of this work. An alternative is to reduce the number of cores contained in the narrow SLR beyond the proportional reduction in our current floorplan, but this is not possible with Manticore's current design: cores must be adjacent to their NoC switch, so moving cores out of the narrow SLR moves the switches with them. This further lengthens the distance between switches in the top and bottom SLRs, resulting in low clock frequencies.

The next section describes a minor change to Manticore's design that decouples a core's placement from that of its switch, allowing us to implement a superior floorplan.

Irregular Split Grid

The top part of Figure 4.8 gives a high-level view of our desired floorplan. The key idea is that links between NoC switches cannot cross an SLR boundary at high clock frequencies[4], and so all switches should be placed in a single SLR. We reserve the narrow SLR for this purpose: since switches use only CLB resources (which are abundant), they will not cluster around the URAM columns in the narrow SLR, freeing up the cramped space around the shell and enabling short connections between neighboring switches. We then partition cores equally between the full-width top and bottom SLRs on the device. One exception is the privileged core, which we leave in the narrow SLR as it must be close to the DRAM cache and to Manticore's controller.

Implementing the above floorplan requires decoupling a core's placement from that of its NoC switch. Figure 4.9 shows how we modify each core to do so.

Egress We pull the core's NoC egress path as early as possible in its pipeline (early in the execute stage). We then push the 7 pipeline registers that were previously *inside* the core (between the decode stage and the NoC egress path) to be *outside* the core. This gives a signal emitted from a core 7 cycles of latency to traverse the FPGA, cross the SLR boundary, and reach its NoC switch in the central SLR. Our modifications to the NoC egress path are

[4] We explore alternative NoC microarchitectures that would not have this issue in Section 4.9.

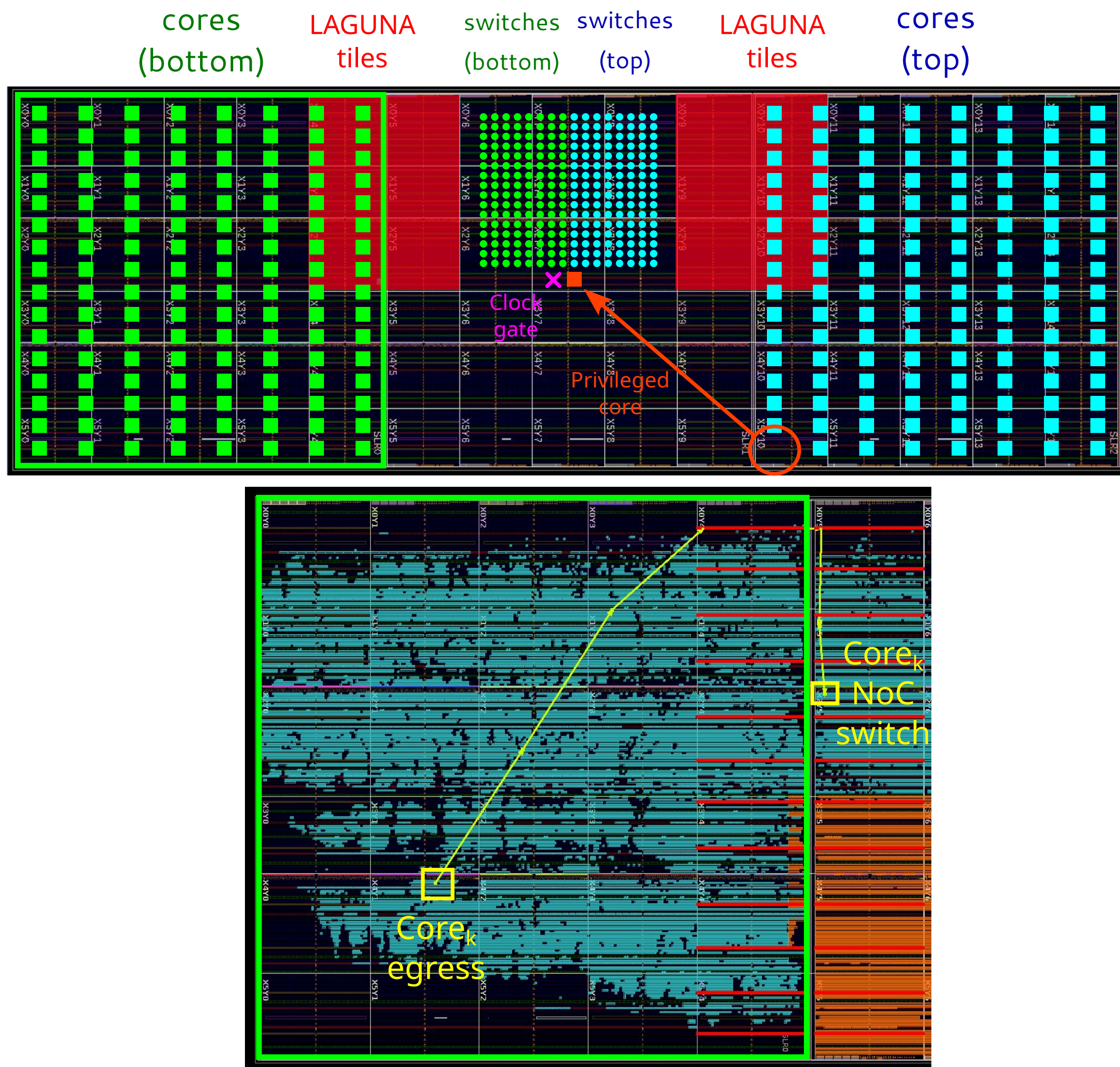


Figure 4.8 – Irregular split grid floorplan. The top part shows an abstract representation of where cores and switches are placed. The bottom part shows how, in practice, the hard LAGUNA registers are used to cross SLR boundaries at high speed. A core sends a message through its NoC egress port, which takes multiple cycles to reach a LAGUNA tile. The LAGUNA tile forwards the message to the adjacent SLR where standard pipeline registers then continue routing the message to its NoC switch. Note that LAGUNA tiles are present at regular intervals along the full SLR boundary width, but Manticore can use only the LAGUNA tiles adjacent to the narrow portion of SLR1 as those in the shell are inaccessible.

---

invisible to the compiler as it still has the same latency—we simply moved existing pipeline registers around.

Ingress The NoC ingress path must also be pipelined to decouple the placement of the NoC switch from that of the core. However, changes to the NoC ingress path introduce new pipeline registers, so we modify the compiler's scheduler to add 7 cycles of latency to the arrival time of NoC messages.

We now tackle the issue of longer wire delays in SLR crossings. UltraScale+ FPGAs contain special hard pipeline registers (called LAGUNA registers) to help cross SLR boundaries at high

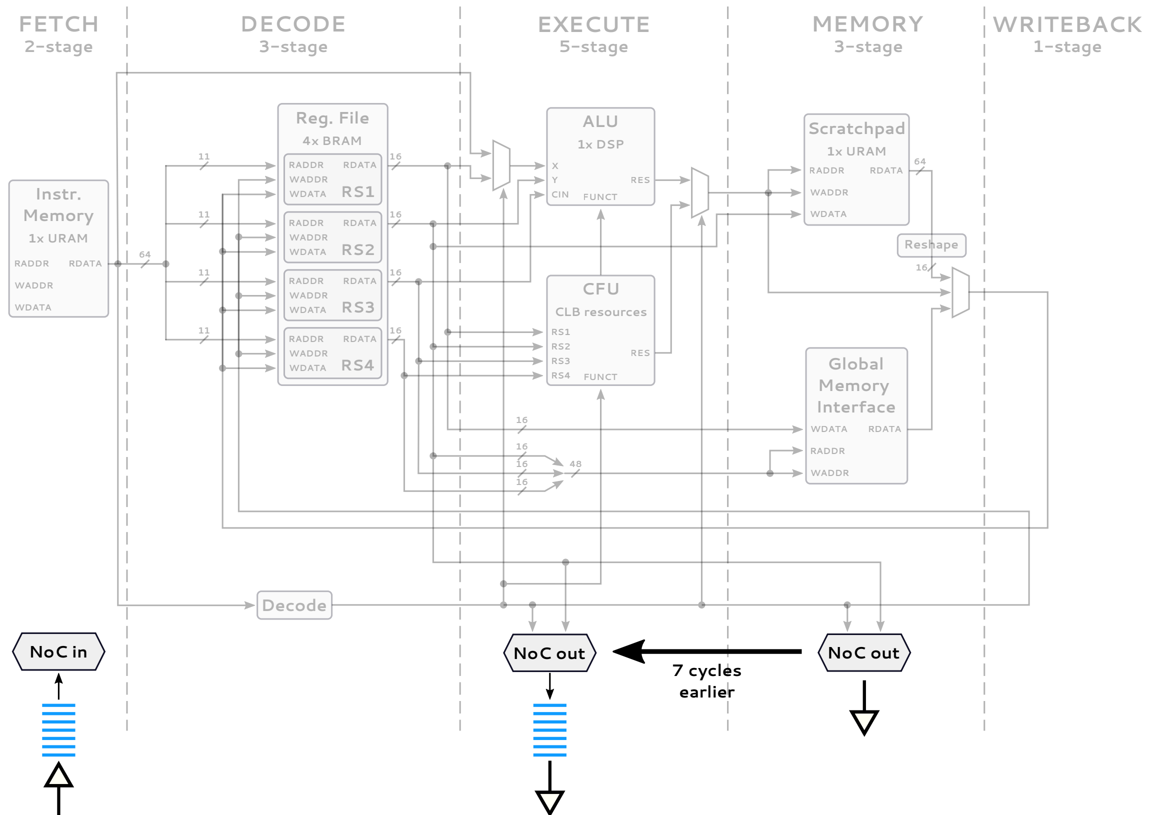


Figure 4.9 – Decoupling core placement from NoC switch placement. We move the core's NoC egress port to be earlier in the pipeline and push the intermediate pipeline registers (blue bars) outside the core. We then add 7 pipeline registers outside a core's NoC ingress port to ensure a core can be placed far from its switch.

clock frequencies (see Figure 4.8). Using LAGUNA registers requires populating two registers in opposing LAGUNA tiles: one TX register in the source SLR, and one RX register in the destination SLR. The TX → RX path is a direct link between two registers, so there must be no logic between two user registers for them to be mapped to the LAGUNA registers. The now-pipelined core → switch and switch → core paths are composed entirely of back-to-back pipeline registers, and so are perfect candidates to use LAGUNA registers to cross SLR boundaries at high speeds.

Note that this floorplan simply constrains cores to either the top or bottom SLRs, not to specific rows within them. Vivado is therefore free to place cores arbitrarily, which results in a highly irregular floorplan with unintuitive paths between cores and their NoC switches. For example, the bottom part of Figure 4.8 shows a case where a core located on the right side of the device ends up sending messages diagonally through the device to its left-most LAGUNA tile, before crossing rightwards to its NoC switch. However, the 7-cycle latency between cores and their NoC switch is long enough for such paths to exist without harming achievable clock frequencies. In fact, we tried further constraining the design to enforce that cores and their NoC switches be placed in similar quadrants, but it had little effect on the final clock frequency (we were limited by issues either inside the cores, or in Manticore's control domain due to the stochastic nature of P&R).

Figure 4.10 – Enforcing relative placement of BRAMs and URAMs in cores. We show the register files and instruction/data memories of four cores. Cells with the same color belong to the same core. BRAMs are enforced to be placed in always the same order (RS1, RS2, RS3, RS4; from top to bottom). A similar strategy is used to enforce that the instruction memory always appear above the data memory (this is less necessary, but we found it leads to more regular local floorplans if enforced).

This concludes our high-level floorplanning efforts. What remains is to handle lower-level issues inside the cores.

4.8.3 Low-level Floorplanning

P&R is known to be a noisy process as tiny design changes can lead to large QoR differences between design runs. In particular, P&R tools do not have a human's perception of how certain logic structures in a design should be grouped for good performance, especially when components that should be physically adjacent are in different hierarchies. In Manticore's case, Vivado often places the four URAM banks that comprise Manticore's cache in *different* URAM resource columns. A similar problem occurs for the four BRAMs that comprise a core's register file, whose placement in different resource columns causes excessive local routing.

We overcome this issue by using relative location constraints in Vivado to force adjacent placement of BRAMs and URAMs in the same core and in the cache (see Figure 4.10). A relative location constraint enforces a local XY displacement between named entities during placement. These constraints consistently led to better floorplans and, consequently, to better clock frequencies. Note that we chose not to use any floorplanning for the CFU as it is formed of far too many

CLBs. Constraints are best applied to individual large structures (e.g., BRAMs and URAMs), not hundreds of small ones (e.g., CLBs).

4.8.4 Floorplanning Results

Figure 4.11 compares Vivado's automatic floorplans with our final irregular split floorplans, which allows large 15×15 designs to run at 475 MHz, and 16×16 designs to run at 450 MHz. We are confident more floorplanning would bring performance up to 500 MHz, but we decided that it was not worth the extra effort and we froze the design at this stage to better focus on the compiler.

4.9 Alternative Microarchitectures

Manticore is an *architecture* that supports the fine-grain parallelism in RTL simulation. A *microarchitecture* demonstrates the strength of an architecture through a specific implementation. This chapter presented a microarchitecture for Manticore that is suitable for an FPGA-based implementation. Naturally an ASIC-based implementation does not have the same constraints as an FPGA-based one, so Manticore's microarchitecture might look very different in such a setting.

Exploring different microarchitectures is orthogonal to Manticore's end goal of demonstrating that, with proper architectural support, we can reach simulation speeds that are unattainable on a general-purpose architecture irrespective of how optimized its software is. Nevertheless, we present some alternatives that we considered for our FPGA prototype and discuss why we did not pursue them.

4.9.1 Core Designs

Our current core uses a simple single-issue design. There is considerable parallelism in RTL simulation and a multi-issue core would certainly reduce the computational critical path. However, a multi-issue processor requires a register file with multiple write ports. Multi-ported register files can be implemented by using additional resources (LVT- or XOR-based), or time-multiplexing (multi-pumping).

LaForest et al. [61] present an in-depth analysis of multi-ported register file implementations using logic elements, live value tables (LVT), or XOR-based approaches for different port counts (2W4R, 4W8R, 8W16R). We ignore logic element-based implementations as they are infeasible for large register files that support on the order of 1k entries. Table 4.3 reproduces reported numbers for a 1024-entry register file as it is the closest evaluated configuration to Manticore's large register file. The table also reports the *theoretical* maximum number of cores that we could fit on the U200 given the BRAM requirements of each configuration. The minimum number of BRAMs needed to implement a single 1024-entry 2W4R register file is of 8, which severely limits the number of cores we would be able to fit on our FPGA. Implementing a multi-ported memory using these methods is feasible for a small number of high performance processors, but it does

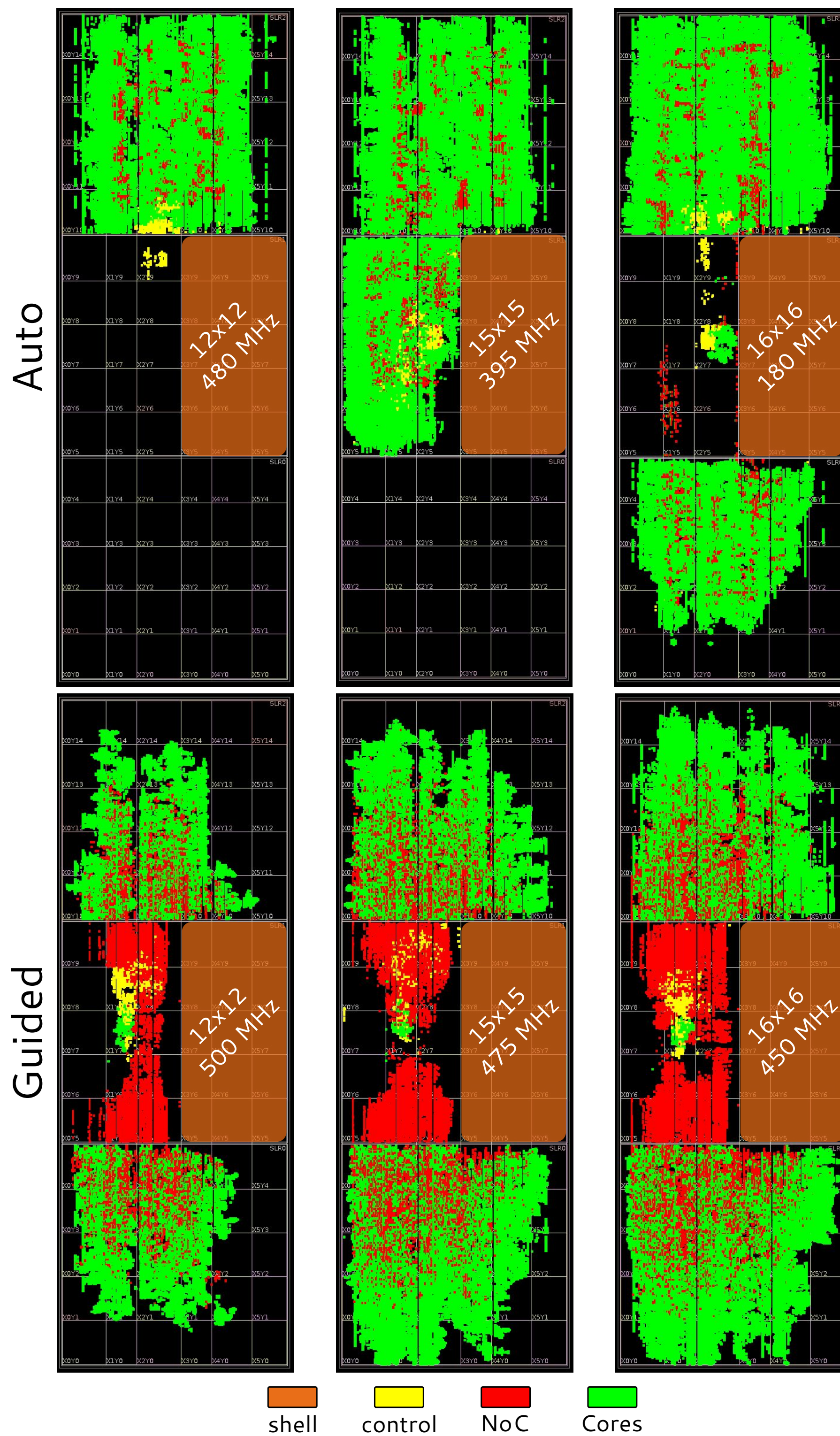


Figure 4.11 – Automatic vs. guided floorplanning. The singular green spot in the narrow SLR with guided floorplanning is the privileged core; all other cores are equally partitioned between the top and bottom SLRs. NoC switches (red, center) are entirely placed within the narrow SLR. Red regions in the top and bottom SLR represent pipeline registers between cores and their switches.

Configuration	LVT				XOR			
	Fmax	Slice	BRAM	# cores	Fmax	Slice	BRAM	# cores
2W4R	294	1655	8	**232**	299	60	10	**186**
4W8R	213	5644	32	58	238	289	44	42
8W16R	104	20882	128	14	136	501	184	10

Table 4.3 – Resources needed to implement 1024-entry multi-ported register file. LVT-based and XOR-based memories are described in detail by LaForest et al. [61]. All numbers are reported for the Virtex-6 XC6VHX380T device. We estimate the theoretical maximum number of cores than can be implemented on the U200's FPGA given the reported BRAM numbers.

not scale to the hundreds of processors in the Manticore architecture. We would also need a more extensive re-design of the core's pipeline to achieve high clock frequencies when reading the register file.

Multi-ported memories can also be implemented using multi-pumping. Multi-pumping multiplies the number of read and write ports on a memory by internally clocking the memory at a multiple of the external clock and time-multiplexing access to its single native read and write port. Multi-pumping has the advantage that it does not require any additional memory resources. Assuming a multi-pumping factor of two (which is the lowest possible), then each 1W1R BRAM turns into a 2W2R memory. We need at least 4 read ports to implement binary arithmetic, so using 2 BRAMs would yield a 4W4R register file. This is half the number of BRAMs that our current core uses (if we sacrifice the CFU) and could allow us to double the number of cores on the device. However, in reality we are still limited by the URAM capacity of the U200, which means no additional cores can be instantiated. We could have mitigated this issue by making cores heterogeneous such that some have data memories and other do not, but this would complicate our compiler, so we ruled it out.

Finally, multi-pumping is suitable only when the external circuit frequency is low compared to the maximum supported frequency of FPGA primitives. Our current prototype is not in this region of the design space as our external frequency is already close to the maximum that is achievable on an FPGA (≈500 MHz). If we were to multi-pump the register file by a factor of two, we would need to reduce the overall operating frequency of the design to ≈250 MHz. Compiling for a dual-issue processor that runs at half the frequency of a single-issue processor would not yield any performance benefit (but it would simplify floorplanning).

We have only performed a high-level analysis at this stage, but more subtle details also need to be handled: a multi-issue processor needs an even wider instruction memory. Assuming we drop the CFU in each core, we computed that we would not be able to bundle two instructions in a 72-bit URAM word given the large number of bits needed to index each core's large register file. We considered compressing the instruction space by using a base plus offset-based register indexing scheme, but it would needlessly complicate the job of the compiler and we decided not to pursue this option. An alternative solution would be to use a smaller register file. However, we could not evaluate the performance impact of a smaller register file at the time since the

compiler was built in parallel to the hardware and was incomplete, so we decided to stick with a very large register file. We will revisit this choice in the evaluation (Section 6.6.4).

Finally, a multi-issue processor is not very useful if the NoC can send only one message per cycle, especially towards the end of a simulated cycle when most cores are communicating their register updates to others. We would therefore need to widen the NoC interface, which would further complicate floorplanning.

In summary, we considered designing a multiple-issue processor, but ruled it out as it was unclear whether it would increase performance due to (1) the reduction in number of processors, (2) the reduction in clock frequency, and (3) the need to re-engineer extensive parts of the microarchitecture to avoid bottlenecking the core's pipeline.

4.9.2 NoC Designs

The design space of a NoC is very large and we did not try to explore all options.

NoC topologies with multiple paths between cores (e.g., folded clos) would permit communication that would otherwise be impossible if only a single link is available and is busy. Such NoC topologies are more complicated to floorplan than Manticore's simple 2D torus, but not by a large margin, and should have been considered more seriously early in the design process.

We considered augmenting the NoC with a broadcast mechanism to reduce the number of SEND instructions needed when communicating high-fanout wires between program partitions. Broadcast benefits signals that have *very* high fanout (e.g., global resets, etc.). Good designs typically do not have such signals as they result in low clock frequencies and so use either hierarchical alternatives or reset only what explicitly needs to be reset, greatly reducing fanout. However, should a design nevertheless contain a high-fanout signal, then broadcast support would certainly help Manticore's performance.

Manticore's switch → switch links cannot cross SLR boundaries at high clock frequencies as their design is incompatible with those of LAGUNA tiles, which require user designs to expose two back-to-back registers without fanout to be mapped to LAGUNA registers. Manticore's switches are one hop away from each other, and so there are no back-to-back registers. One solution would be to pipeline the links between NoC switches such that each link contains at least 2 registers. However, doing so would double the latency of all NoC traffic and its effect on performance was unclear. We also did not want to change the compiler's NoC model given our constraints at the time, and so preferred a floorplanning-only solution that required adding a constant (7 clock cycles, see Section 4.8.2) in only the compiler's instruction scheduler. Nevertheless, had we pipelined the switch → switch paths, it would make sense only if switches remain close to their core so that the design can remain as regular as possible. Doing so, however, means that cores would end up in the narrow SLR, which is already congested due to the scarcity of URAMs at the left-most side of the FPGA.

In summary, the NoC implementation we pursued certainly had limitations, but it fit the constraints imposed upon us by our FPGA.

4.10 Summary

We presented an FPGA-based implementation of the Manticore architecture on a large, datacenter FPGA. Cores consist of deep, feed-forward pipelines to achieve a high clock frequency. They support standard binary arithmetic and 32 custom functions that can implement any 4-input bitwise logic expression. Cores are interconnected with a simple NoC that is buffer-less and uses dimension-ordered routing. Cores can perform only blind writes to remote cores over the NoC.

The Manticore architecture provides a deterministic machine to enable device-wide static scheduling. We use clock gating on the FPGA as the mechanism for maintaining determinism when interacting with logic that has non-deterministic latency (off-chip DRAM access and the host x86 processor). An on-chip cache intercepts traffic to off-chip DRAM to absorb its high latency during repeated accesses.

While Manticore's architecture presents itself as a regular grid, its physical implementation on our FPGA uses an irregular layout to work around imbalances in resource distribution across the device and achieve a high clock frequency. Our fastest configuration is a 15×15, 225-core 475 MHz Manticore design.

This concludes our presentation of Manticore's implementation. Chapter 5 presents Manticore's compiler.

5 Manticore Compiler

Chapter 4 described Manticore's hardware, in particular its deterministic behavior that permits the implementation of a static BSP execution model. This chapter now describes Manticore's compiler, which uses the hardware's determinism to schedule RTL code to instructions that Manticore's hundreds of cores can efficiently execute. Manticore's deterministic hardware lacks interlocks and buffering, so it relies *entirely* on its compiler to parallelize the input netlist, schedule instructions within each core to avoid data hazards, and schedule messages between cores to avoid structural hazards on Manticore's NoC.

5.1 Overview

Figure 5.1 presents an overview of the RTL-to-instruction mapping process. Compilation is decomposed in two stages: a Verilog frontend and Manticore's backend compiler. The frontend parses an RTL design expressed in Verilog, converts it into a netlist DAG, then emits a single, large basic block of code in Manticore assembly language (MASM) that executes the DAG. The backend operates on two related IRs: (1) netlist assembly, and (2) lower assembly. Netlist assembly is basically an IR that supports operands of arbitrary bitwidth, whereas lower assembly is closer to Manticore's ISA and supports only 16-bit operands. Both IRs use static single-assignment (SSA) and can be interpreted in software. Lower assembly's software interpreter is a full-fledged ISA simulator that is parameterized by Manticore's hardware configuration. We use the interpreters extensively to validate the compiler's passes.

We derived our Verilog frontend from that of Yosys [113], which is an open-source framework for RTL synthesis. Yosys supports multiple synthesis algorithms which can be combined into flows using scripts. Custom passes can also be designed by directly extending Yosys' C++ codebase. We extended Yosys to support basic system calls, such as `$display` and `$finish`, required for simulation. After parsing the Verilog input, the frontend flattens the design hierarchy, performs a few optimizations, and emits netlist assembly. Because of the semantics of RTL code, instructions in netlist assembly are unordered and have arbitrary-width operands.

The backend orders the instructions and applies simple optimizations (dead code elimination, constant folding, and common subexpression elimination). We then transform the netlist assembly

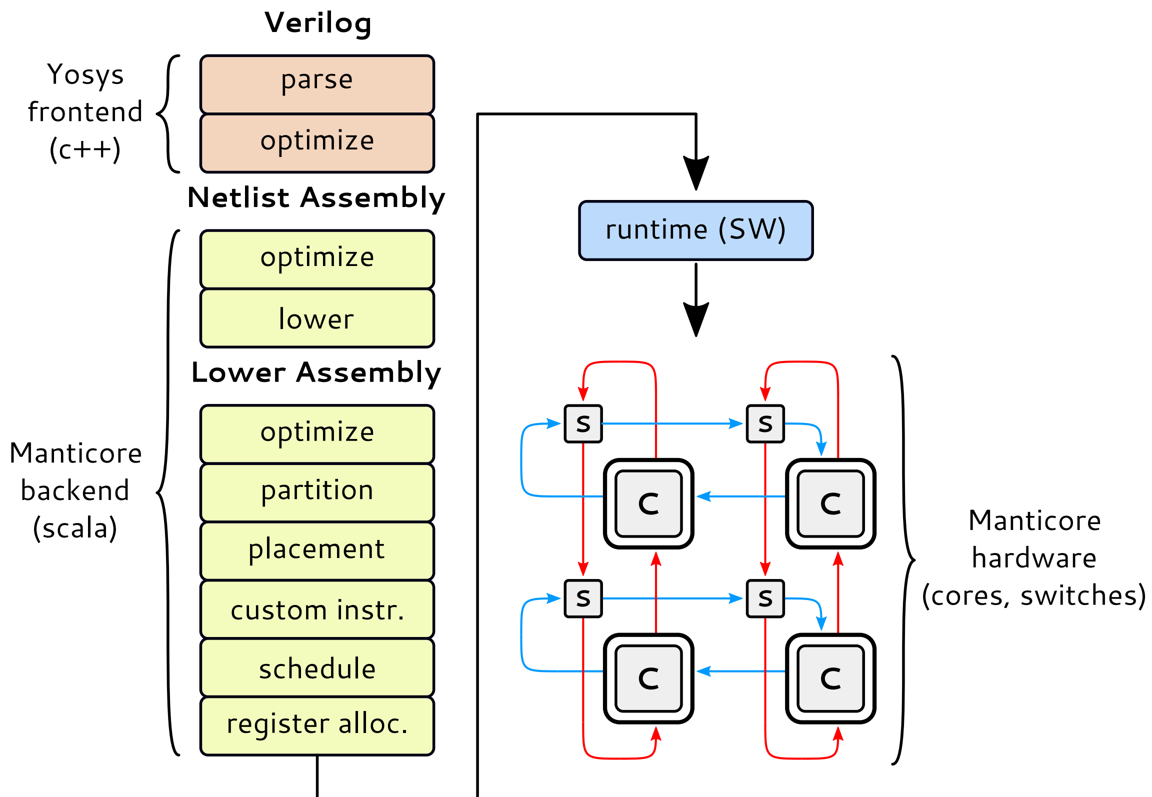


Figure 5.1 – Compilation overview. Compiling a Verilog design down to instructions for a Manticore processor is a two-stage process: (1) A frontend parses and elaborates the Verilog design, performs some synthesis optimizations, then emits a long basic block of code that executes a netlist DAG. The DAG is described in *netlist assembly*, the backend's high-level IR that supports arbitrary-width operands (similar to Verilog wires that can be arbitrary-width). (2) The backend then optimizes the netlist assembly and lowers it to *lower assembly*, where all operands are 16 bits wide. The backend then parallelizes the code, assigs each partition to a specific core, synthesizes custom functions in each partition, and globally schedules instructions across all cores. The output of the compiler is a set of binaries that are loaded onto a Manticore processor (bottom-right) by a runtime running on a host x86 processor.

---

instructions into an equivalent sequence of lower assembly instructions whose operands match Manticore's 16-bit data path. Initially, the lower assembly is a monolithic sequence of instructions (a single process). After further optimizations, the compiler partitions the instructions into multiple processes and assigns each process to a core in the Manticore grid. The compiler then optimizes each process by fusing chains of bitwise logic instructions into custom instructions.

The final steps of compilation are scheduling and register allocation. Scheduling ensures that there are no data hazards in the pipeline by inserting `NOP` instructions to enforce data dependencies. In addition, the `SEND` instructions are scheduled to ensure timely message delivery. The compiler then maps virtual registers to machine registers and emits binary code.

The Yosys Verilog frontend passes are roughly 2K lines of C++. The Manticore backend compiler is 18K lines of Scala. The runtime is built on top of the Xilinx runtime library (XRT) with around 800 lines of C++ code.

5.1.1 Note on Program Optimizations

Manticore's frontend uses many of Yosys' built-in synthesis optimization routines. We generally select the subset of optimizations that reduce circuit area as Manticore's cores have limited instruction memory.

Manticore's backend compiler is written from scratch in Scala and took a significant portion of our total effort to get correct so it could produce functional Manticore programs. We limited ourselves to implementing only the most basic optimizations that are available in standard compilers (alias removal, dead code elimination, constant folding, and a simplified version of common subexpression elimination).

The backend views input Verilog designs as a *black box*, i.e., it assumes nothing about the designs other than the fact that they are single-clocked. Higher-level knowledge of circuit topology (systolic, etc.) could help develop more sophisticated partitioning and placement strategies; we chose to focus on automated techniques that work on any circuit. The compiler's generated code therefore represents a lower bound on achievable performance as many optimizations are left unexplored.

5.2 Frontend

5.2.1 Terminology

We first describe some terminology that Yosys uses before expanding on the frontend's operation.

Yosys represents circuit elements as *cells*. Cells represent basic logic gates (e.g., `$and`, `$xor`, etc.) or compound gates that encapsulate complex functionality (e.g., `$sdffe` is a D flip-flop with synchronous reset and an enable signal). Cells exist for large RTL constructs such as RTL memories.

Cells are interconnected using *signals*. A signal represents a concatenation of multiple primary building blocks into a wide unnamed wire. A signal's building blocks are (1) a full-width wire, (2) a subset of a wire, and (3) a constant.

5.2.2 Compiling RTL to Netlist Assembly

Yosys is a *synthesis* tool that operates on designs described in Verilog-2005. Its parser therefore understands only the synthesizeable subset of Verilog-2005 and various language features related to formal verification. However, Manticore is an accelerator for RTL *simulation*, so we need support for some non-synthesizeable Verilog keywords that are used to support basic system calls during simulation (e.g., `$display`, `$finish`). We modify Yosys' frontend to add support for these keywords, which we hoist out of nested procedural statements and into a custom Yosys cell at the root of the module in which they are found. This cell is guarded with a compound logic expression (since it was hoisted out of a nested procedural statement).

We then elaborate the design hierarchy and call Yosys' behavioral synthesis script to convert behavioral Verilog processes (containing `if` or case statements) into a structural netlist that consists entirely of registers, memories, and wires that connect them. Note that RTL languages like Chisel directly generate *structural* Verilog, but hand-written Verilog designs are often behavioral and need to undergo behavioral synthesis to become a netlist. Yosys' ability to perform behavioral synthesis was one of the original reasons why we chose it as our frontend. A structural netlist does not contain branch conditions, and so is a good fit for Manticore's branch-free execution model. Nevertheless, Manticore does require support for predicated instructions to implement other functionality (e.g., selectively writing to RTL memories, etc.).

Next we apply several of Yosys' built-in optimization passes. Manticore's instructions all have the same latency, so shorter assembly programs are faster than longer ones. We therefore select Yosys' synthesis optimizations that reduce circuit area. A byproduct of Yosys' optimizations is the introduction of custom cells to represent registers with a combination of different feature sets (e.g., sync/async reset, set/clear signal, enable signal, etc.). We revert these registers back into plain registers and a series of multiplexers to be compatible with Manticore's simple instruction set.

Yosys' internal representation is adequate for a synthesis or a place-and-route tool that handles bit-level logic, but is at times poorly suited for generating assembly code. The last few stages of the frontend therefore consist in multiple custom passes to clean up Yosys' representation into one more suitable for code generation. Using a simulator's frontend would have led to much better sequential code generation, but we decided to stick with Yosys as it allows us to obtain a structural netlist with ease—something that is not trivial to extract from a software simulator.

- Verilog supports subword assignment semantics (different bits/chunks of a wire can be assigned in different places), which makes emitting SSA assembly challenging. We develop custom Yosys passes to group all writes to a signal in a single place to remove subword assignment operations.
- Yosys' cells generally require operands to be of equal width, so it automatically pads the widths of Verilog signals before connecting them to target cells. If a wire represents a signed Verilog value, Yosys performs the sign extension by a sequence of 1-bit concatenations, which needlessly increases code size. These patterns appear surprisingly often in Verilog code, so we detect and replace them with a compact equivalent which is representable by a much shorter sequence of assembly instructions.

Finally, we emit a *monolithic* basic block of MASM code in the high-level *netlist assembly* IR, which we pass to Manticore's backend compiler. Because of the semantics of RTL code, instructions in netlist assembly are *unordered* and have arbitrary-width operands.

5.3 Backend

The compiler must solve three main problems to produce a functional Manticore program: *partitioning* (what to compute?), *placement* (where to compute?), and *scheduling* (when to compute?).

Partitioning decomposes a single, monolithic basic block of instructions into independent program partitions. The goal is to distribute work among *logical processes* such that each process is balanced (contains roughly the same number of instructions). The partitioner produces at most as many processes as there are cores in a Manticore grid.

Placement assigns each logical process to a *physical core* in a Manticore grid. The goal is to reduce NoC traffic between cores. Each core is responsible for the execution of *at most* one process (i.e., some cores can be unused).

Scheduling globally schedules instructions in all cores. The goal is to avoid data hazards in the cores' pipelines and ensure conflict-free use of the shared NoC by all cores.

Unfortunately these problems are co-dependent:

- Two balanced partitionings of the same program may greatly differ in the amount of inter-process communication they exhibit at runtime over a shared NoC, thereby affecting performance. It is therefore not enough for the processes to be balanced, but we also want a partitioning algorithm that reduces inter-process communication. Partitioning therefore requires an estimate of communication costs between processes, but this cost is known only after placement when the distance between the cores are known.
- Placement requires knowledge of NoC link traffic to best place processes, but link traffic is known only after scheduling. However, scheduling cannot occur before placement as the compiler needs to know the final assignment of processes to cores in order to schedule NoC traffic.

Partitioning is an instance of a general graph partitioning problem, which is NP-Complete [16] and cannot be optimally solved in a tractable manner, so we must rely on heuristics instead.

Placement and scheduling in spatial architectures are co-dependent, but have been shown to be optimally solvable. Nowatzki et al. [70] provide an in-depth analysis of using mixed-integer linear programming (MILP) to formulate and optimally solve placement and scheduling problems through five abstractions: placement of computation, routing of data, managing event timing, managing resource utilization, and forming optimization objectives. The main idea is to map a compute DAG G onto a hardware graph H, which is demonstrated through a rich collection of software benchmarks and spatial architectures (TRIPS [18], DySER [41], and PLUG [21]). The proposed MILP formulations are solved in durations that range between 1 s and 6 min. However, these benchmarks represent succint, loop-based code that are encoded as a graph G that contains at most few tens of vertices (max 40). By contrast, Manticore's program graphs (after partitioning) and can contain as many vertices as there are cores, which is over 200 in our small-scale prototype. Given the execution time of the proposed MILP formulation on small programs and the $\approx 5\times$ larger size of Manticore programs, it is likely that jointly solving placement and scheduling is intractable, so we choose to solve each step separately.

We use the Verilog snippet in Listing 5.1 as a running example to illustrate partitioning and placement in the following sections. Note that the circuit does not compute anything meaningful; it is simply small enough to demonstrate the compiler's functionality.

```verilog
module paper_circuit (
  input wire clock
);
  reg [15:0] R1, R2, R3, R4, R5;
  wire [15:0] L1, L2, L3, L4, L5, L6;

  assign L1 = R1 + (R5 & L2);
  assign L2 = R1 - R5;
  assign L3 = R1 * R2;
  assign L4 = R3 == 0;
  assign L5 = R3 | R4;
  assign L6 = R4 ^ L5;

  always @(posedge clock) begin
    R1 <= L1;
    R2 <= L2;
    R3 <= L3;
    R4 <= L4;
    R5 <= L6;
    $display("R1 = %d", R1);
  end
endmodule
```

Listing 5.1 – Verilog module used as a running example for partitioning and placement. This is the same example as the abstract DAG shown in Figure 3.1.

5.3.1 Partitioning

Partitioning instructions across the cores is the most critical step to achieving good parallel performance. Despite the absence of runtime synchronization in Manticore, data movement is still costly and excessive communication will limit scalability. The partitioner's goal is to reduce communication between cores, while distributing work as equally as possible to avoid stragglers. Since partitioning occurs before placement, the compiler has no knowledge of the cores on which computation is mapped. We therefore use the term *process* here instead of "core" to refer to a *logical* collection of instructions that are executed in a single core.

We perform partitioning in a bottom-up manner (i.e., we disregard information about the RTL hierarchy). We chose to do so explicitly as RTL hierarchies are virtual: two deeply-nested combinational gates in different RTL design hierarchies may have direct wires between them, which a hierarchical representation does not capture. In this case, if we place logic from separate hierarchies in different processes, then we would have an invalid partition: Manticore's BSP model of computation allows communicating only registers between processes, but here we would need to communicate combinational wire values.

Figure 5.2 outlines the partitioning process. The compiler parallelizes a single, monolithic assembly process into a *process graph* (bottom of Figure 5.2) in two steps:

1. *Split* the monolithic process into a maximal number of tiny processes.

2. *Merge* the split processes so that the total number of processes does not exceed the number of available cores.

Split

The compiler first creates a DAG representing data dependencies in the monolithic process. It then uses a backward traversal to partition the vertices reachable from each data sink into independent smaller processes (top of Figure 5.2). All DAG vertices that are used to compute a register's next value (<name>+ in Figure 5.2) are duplicated (not shown), maximizing parallelism at the expense of increased computation. However, the compiler ensures that instructions that access the same memory region (e.g., an unpacked array in Verilog) end up in the same process to avoid moving large amounts of data at each RTL cycle. In addition, all privileged instructions must execute in the same process (instructions in yellow in Figure 5.2).

The example in Figure 5.2 results in 8 maximally-parallel processes after the splitting stage.

Merge

If we view the maximal set of split processes as a graph whose vertices denote processes and edges denote communication, then merging is a graph partitioning problem. Existing partitioning tools [55, 85] assume a *linear* cost function: merging two vertices A and B produces a vertex C with weight $W_A + W_B$. However, optimizations such as data sharing and duplicate code elimination make merging non-linear in Manticore, so we develop a custom heuristic algorithm instead. Duplicate code elimination during merging is important in Manticore as each core has limited instruction space (at most 4096 instructions). Merging can continue even after reaching the number of available cores because it can reduce execution time. For instance, merging processes `A` and `B` that read a value produced by process `C` could lower the execution time of `C` because it executes one fewer `SEND` instruction.

Since partitioning is performed before scheduling, the compiler estimates the execution time of a process as the total number of instructions it executes, including `SEND`s, but excluding the `NOP`s used to schedule data hazards and received messages (`NOP`s are known only after scheduling). A vital goal of the merging stage is to avoid forming stragglers by equalizing processes' their execution time. The compiler iteratively picks two merge candidates that minimize the increase in merged execution time. It starts from the process with the shortest execution time and merges it with another process with which it communicates. Intuitively, by starting from the smallest processes and constructing larger ones, we can balance the execution time of the processes and simultaneously reduce communication (hence reducing network contention).

On the example in Figure 5.2, the partitioning algorithm produces a *process graph* with four processes at the end of the merging stage.

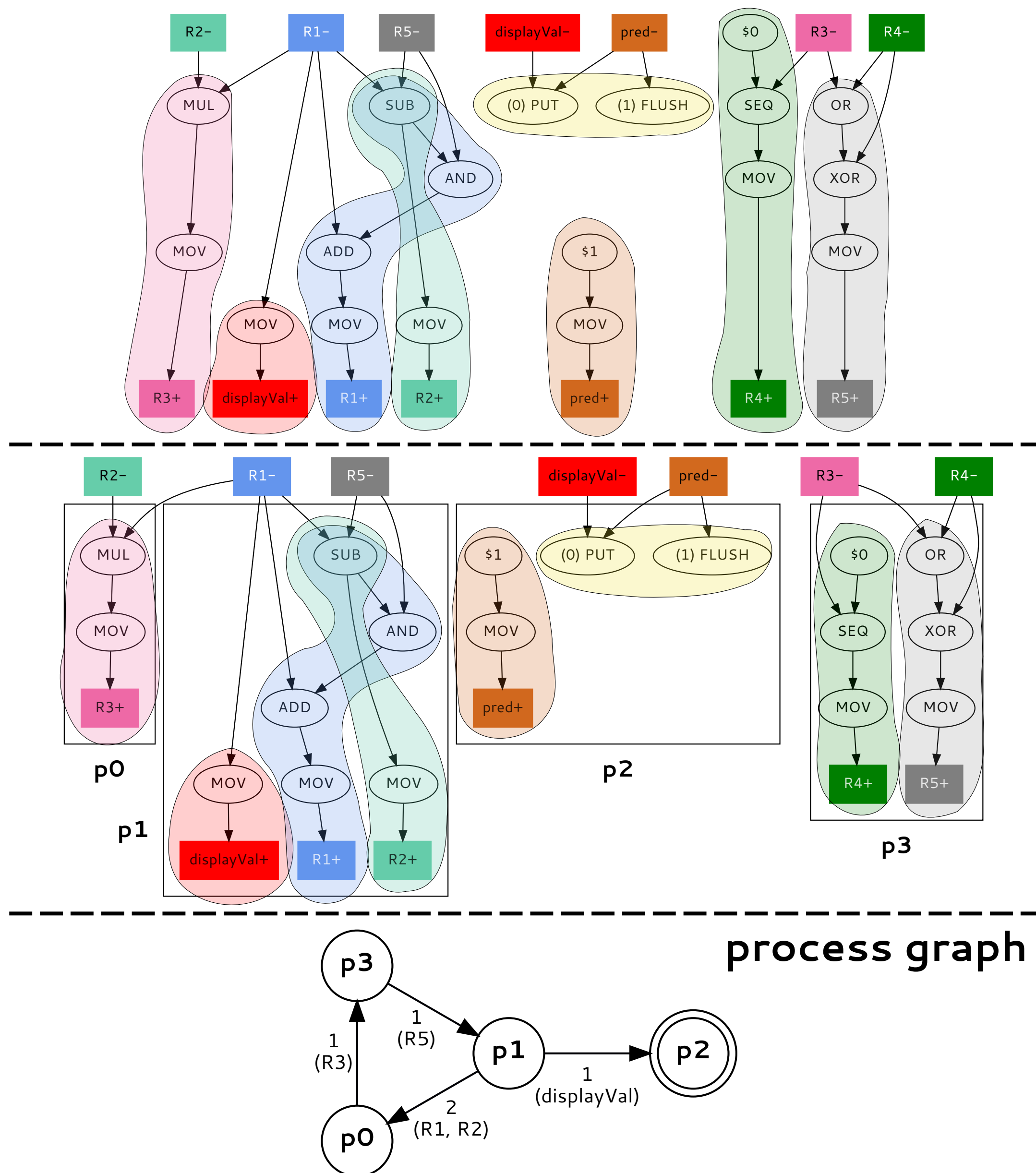


Figure 5.2 – Extracting parallelism through partitioning. We start from the netlist DAG of Listing 5.1 (top), where rectangles at the top and bottom represent current and next register values, respectively. We then *split* the DAG into 8 individual processes that each compute a single next register value (shaded surfaces). In doing so, we duplicate (not shown) all shared intermediate DAG vertices (e.g., SUB) that are used by the generated processes to maximize parallelism at the expense of increased computation. We then *merge* the split processes to produce at most as many processes as the number of cores that are available. This example targets a Manticore grid containing 9 cores, but the partitioner identifies 4 partitions that heuristically result in approximately the same amount of work in each process, including the work needed to send values to other processes. The final output of the partitioner is the *process graph* (bottom) where vertices represent processes and weighted edges represent the number of SEND instructions that are needed between communicating processes. We use a double-circle to highlight the *privileged process*, i.e., the process that contains all privileged instructions.

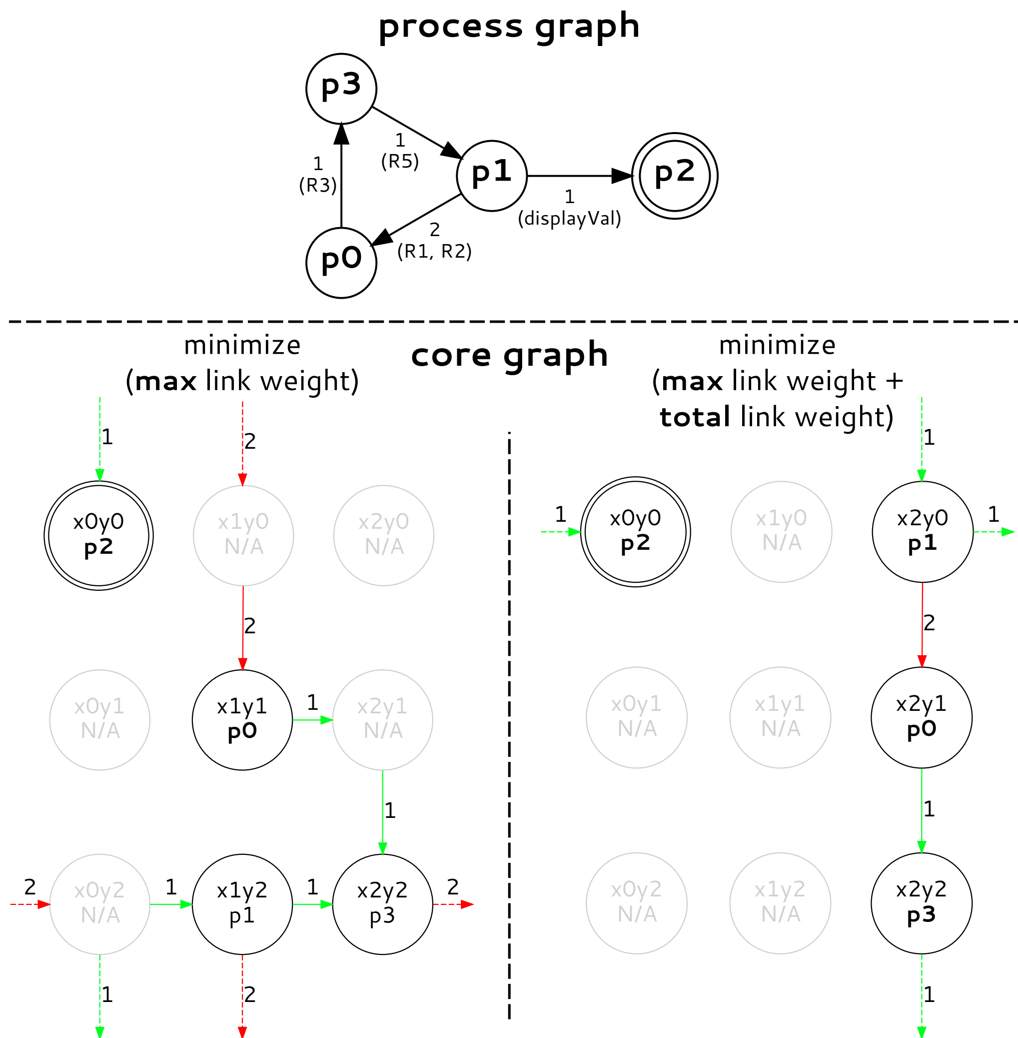


Figure 5.3 – Placing processes on cores. Placement maps vertices in the process graph (top) to vertices in the core graph (bottom). Double-circles correspond to the privileged process and privileged core. We represent the wrap-around links of the torus NoC with dashed lines, i.e., dashed horizontal lines go from the right-most core to the left-most core in each row (similarly for dashed vertical lines). We omit showing links that do not transport any traffic, and gray out cores that are unused. The bottom-left core graph shows link weights if we minimize the maximum link weight, whereas the bottom-right core graph shows link weights if we jointly minimize the maximum link weight with total link traffic.

5.3.2 Placement

Placement is an assignment problem where we map *logical* processes identified during the partitioning stage onto *physical* cores in a Manticore grid (see Figure 5.3 for a 3×3 Manticore grid). Recall that we solve placement pre-scheduling, so we need a heuristic to rank proposed placement strategies. Intuitively, our hypothesis is that reducing aggregate traffic on NoC links would result in a shorter global schedule: if links are less utilized, then the scheduler is less likely to have to delay processes by inserting NOPs to avoid a busy NoC link. We can reduce traffic by assigning communicating processes to nearby cores on the torus NoC.

Mapping processes to cores implies finding a suitable route between cores for each pair of communicating processes. There are an exponential number of paths between two vertices in a general graph, which rules out any optimal assignment through a MILP formulation as it would result in an exponential number of constraints. However, Manticore's NoC is uni-directional and uses dimension-ordered routing, so there is only a single path between any two cores and a MILP formulation entails only a polynomial number of constraints. Given this property, it seems conceivable that we can obtain an optimal solution to the placement problem.

Optimal Assignment

Figure 5.4 shows the MILP formulation we propose to map processes to cores in a Manticore grid. We use two graphs to model the assignment problem: a source *process graph* and a target *core graph*.

The process graph represents logical relationships between processes as a weighted graph $G(P, E, W)$. A vertex $p \in P$ represents a process and an edge $e \in E$ represents directed communication between two processes. Edges are weighted, with $W(e)$ representing the number of `SEND` instructions between communicating processes.

The core graph represents physical relationships between *adjacent* cores as an *unweighted* graph $G(C, L)$. A vertex $c \in C$ represents a core and an edge $l \in L$ represents a physical link between adjacent cores. The MILP solver's task is to compute the link weights and optimize an objective function over them.

The MILP formulation relies on identifying the set R of all possible inter-core routes in the core graph, i.e., routes can span more than one link. There are exactly $|C| * (|C| - 1)$ such routes as each core can send a message to any other core, excluding itself.

To avoid ambiguity, we use the term "edge" to refer to an edge in the process graph, and "link" to refer to an edge in the core graph in the rest of this section.

We define three variables in our formulation:

1. A *binary* variable $x_{p,c}$ for every process and core pair. Variable $x_{p,c} = 1$ if process p is mapped to core c.

$$x_{p,c} \in \{0, 1\} \quad \forall (p, c) \in P \times C$$

2. An auxiliary *binary* variable $y_{e,r}$ for every edge-route pair. Variable $y_{e,r} = 1$ if edge e in the process graph is mapped to route r in the core graph.

$$y_{e,r} \in \{0, 1\} \quad \forall (e, r) \in E \times R$$

3. An auxiliary *integer* variable w_l for every core edge. Variable w_l captures the total number of messages that use physical link l.

$$w_l \in \mathbb{N}_0 \quad \forall l \in L$$

$$
\begin{aligned}
\text{minimize} \quad & \sum_{l \in L} w_l + z \\
\text{subject to} \quad & x_{p,c} \in \{0,1\} && \forall (p,c) \in P \times C, \\
& y_{e,r} \in \{0,1\} && \forall (e,r) \in E \times R, \\
& w_l \in \mathbb{N}_0 && \forall l \in L, \\
& z \in \mathbb{N}_0, \\
& \sum_{c \in C} x_{p,c} = 1 && \forall p \in P, \\
& \sum_{p \in P} x_{p,c} \leq 1 && \forall c \in C, \\
& x_{p,c} = 1 && p = Priv,\ c = Priv, \\
& y_{e,r} = x_{e_{src}, r_{src}} \wedge x_{e_{dst}, r_{dst}} && \forall (e,r) \in E \times R, \\
& w_l = \sum_{e \in E} \left(\sum_{r \in R | l \in r} W(e) * y_{e,r} \right) && \forall l \in L, \\
& z \geq w_l && \forall l \in L
\end{aligned}
$$

Figure 5.4 – Optimal process-to-core assignment MILP formulation.

The MILP formulation does not encode scheduling constraints as placement is agnostic of clock cycles. However, it must encode multiple physical constraints:

1. A process is assigned to exactly 1 core.

$$\sum_{c \in C} x_{p,c} = 1 \quad \forall p \in P$$

2. A core can be assigned at most 1 process as the partitioning algorithm may produce fewer processes than there are cores.

$$\sum_{p \in P} x_{p,c} \leq 1 \quad \forall c \in C$$

3. The privileged process must be mapped to the privileged core.

$$x_{p,c} = 1 \quad p = Priv,\ c = Priv$$

4. There is exactly one route between any two cores due to the NoC's dimension-ordered routing policy. An edge e in the process graph is mapped to a route r in the core graph if (1) the source process of edge e is mapped to the source core of route r, and (2) the destination process of edge e is mapped to the destination core of route r. The $\wedge$ operator in the constraint below is not linear, but can be linearized following standard transformations [28].

$$y_{e,r} = x_{e_{src}, r_{src}} \wedge x_{e_{dst}, r_{dst}} \quad \forall (e,r) \in E \times R$$

5. The weight of a link l is the sum of the weight of all routes r that include link l, weighted by the edges e that use route r.

$$w_l = \sum_{e \in E} \left(\sum_{r \in R | l \in r} W(e) * y_{e,r} \right) \quad \forall l \in L$$

Finally, we must decide on the objective function and its supporting variables. Ideally we want to minimize the *maximum* link weight as it would ensure that no NoC link dominates all traffic. However, there are many possible process-to-core assignments that result in the same maximum link weight, and some of these assignments are objectively worse than others (see bottom-left of Figure 5.3 where the assignment results in odd detours due results in high total link traffic). Another possibility is to minimize the *total* link weights $\sum_{l \in L} w_l$, but this could yield cases where some links are underutilized and a small subset of links are overloaded. Instead, what we really want is to minimize the *sum* of the total link weights and the maximum link weight. This ensures the lowest total link weight, while ensuring that no links become dominant (see bottom-right of Figure 5.3).

1. We introduce an auxiliary *integer* variable z that captures the maximum link weight w_l.

$$z \in \mathbb{N}_0$$

2. The maximum link weight in the core graph is constrained to be larger than all link weights.

$$z \geq w_l \quad \forall l \in L$$

3. The objective function minimizes the sum of link weights and the maximum link weight.

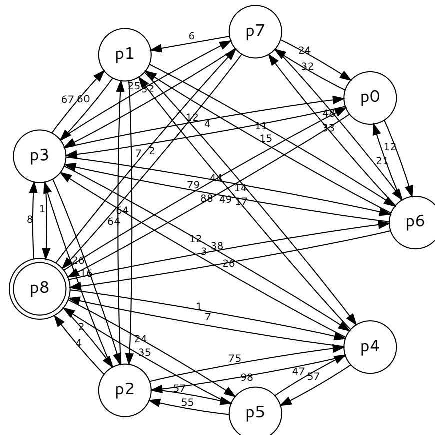
$$\text{minimize} \sum_{l \in L} w_l + z$$

Optimal Assignment Results

The NoC's design turned out to be a double-edged sword: On the one hand it enables a simple, high-frequency physical implementation and a compact MILP encoding that is polynomial in size. However, on the other hand, the MILP formulation for optimizing link utilization is *intractable* for Manticore grids larger than 4×4.

The problem lies in our choice of a uni-directional torus structure as minor changes in the process-to-core assignment can lead to a large difference in link weights. This phenomenon is easier to illustrate on a benchmark that uses all cores in a Manticore grid, so in the rest of this section we use a bitcoin mining benchmark on a 3×3 Manticore grid (see Figure 5.5). Simply swapping the positions of processes `p0` and `p2` leads to a significant change to the weights of *all* links in the core graph, which the solver is forced to compute at each step of its search—an expensive process. Minor changes in the assignment can also lead to large changes in the objective function, which makes the geometry of the search space challenging for a solver to explore.

process graph



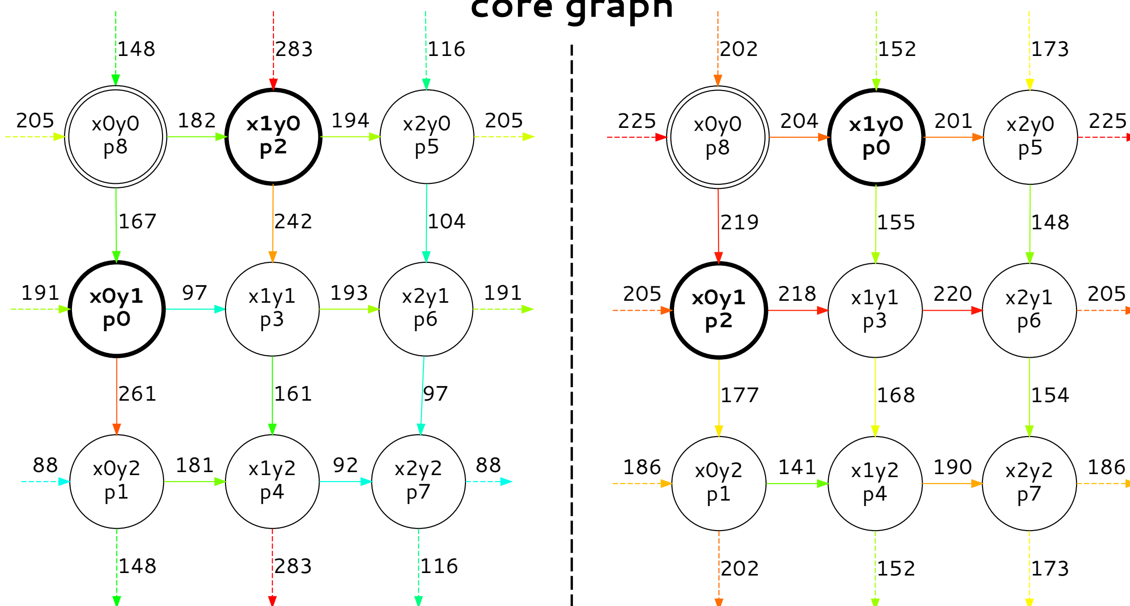


Figure 5.5 – Sensitivity of link weights to minor changes in process-to-core assignment. Minor changes in process-to-core assignment lead to large differences in link weights. We color links by their weight using a heatmap to easily identify hot links (minimum is blue, intermediate is green, high is red). Swapping the core assignments of processes p0 and p2 leads to change of *all* link weights due to the NoC's uni-directional torus structure.

We did not find any obvious structural property to simplify the constraints in the MILP formulation, so we resorted to heuristics for process-to-core placement instead.

Heuristic Assignment

The top part of Figure 5.5 shows the process graph of a bitcoin mining benchmark when constrained to fit on a 3×3 Manticore grid. The process graph has dense connectivity and it is not immediately obvious how to best assign processes to cores, especially in larger process graphs with hundreds of vertices, so we decided to adopt a simple top-down partitioning-based heuristic.

We start with a single cluster containing all processes, which we bi-partition into two equally-sized clusters with the goal of minimizing total communication *across the cut*. We then recursively repeat this bi-partitioning procedure on each sub-cluster until each contains an isolated process. We solve the bi-partitioning problem at each recursive step with a MILP solver, which generally finds a solution in less than a second. Each bi-partitioning stage refines a process' location in a numbered Manticore grid.

Note that the a process' placement following the bi-partitioning procedure does not take into account structural properties of the Manticore grid: processes are simply assigned to one of two subclusters at each recursive stage without any consideration for the placement of the privileged process, which could then be mapped to an unprivileged core—an invalid configuration. However, given the NoC's torus structure, we can simply rotate the process-to-core assignments until the privileged process is placed on the privileged core, which then becomes a valid placement.

5.3.3 Custom Function Synthesis

Custom function synthesis is a peephole optimization that collapses long sequences of *bitwise logic* instructions into a shallow sequence of 4-input custom functions. It is inspired by logic synthesis in a standard silicon flow. We motivate how logic synthesis applies to Manticore programs with the example in Figure 5.6.

The top part of the figure shows a simple Verilog snippet which concatenates four 2-bit wires into a wider 8-bit wire. The bottom-left part of the figure shows the shortest Manticore assembly program needed to emulate the Verilog program. The concatenation requires shifting each wire to the left, masking it with a logical AND to remove high-order bits[1], then using a logical OR to perform the actual concatenation with another wire. Note that this Verilog wiring operation comes *for free* in silicon as it is performed through routing, but costs 9 instructions to emulate in software. The bottom-right part of the figure shows the corresponding Manticore assembly program after custom function synthesis, which collapses the chain of bitwise logic instructions into a single custom function.

Logic synthesis maps a group of RTL gates to a sequence of primitive gates that are available in the target technology. Manticore's CFU is implemented using an array look-up tables (LUTs), the most common primitive in an FPGA. An LUT is a 1-bit, K-input truth table which can represent any logic function of up to K inputs. Modern FPGAs typically contain 6-LUTs, but Manticore's register file supports at most 4 reads per cycle, so the CFU is limited to using 4-input functions. The CFU can implement only logic functions, not arithmetic ones, as we do not use the LUTs' carry chain in our implementation.

We originally envisioned performing logic synthesis in the frontend as Yosys natively supports it. However, Yosys' logic synthesis flow maps most gates (including adders, flip-flops, etc.) to primitive 1-bit gates and breaks the circuit's structure, which results in very inefficient Manticore

[1] Custom function synthesis is performed on lower assembly, i.e., on input programs that consist entirely of 16-bit operations. The compiler cannot—in general—know that the wire represents a narrower 2-bit value, so it cannot elide the AND operation as the wire could be non-zero in its high-order bits.

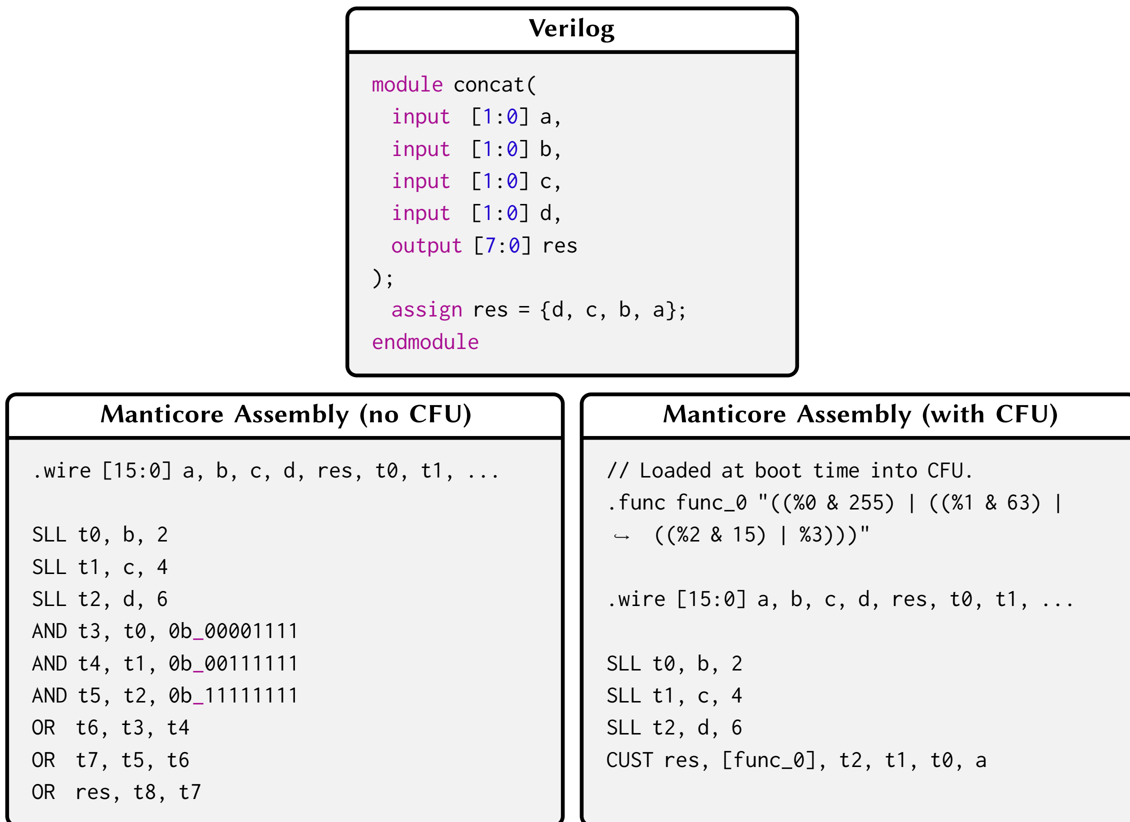


Figure 5.6 – Synthesizing custom functions from Verilog code. The Verilog design (top) simply concatenates four 2-bit wires into a wider 8-bit wire. A minimum of 9 instructions is needed to emulate the circuit without custom functions (bottom-left). With custom functions, the logic component in the dependence graph can be fused into a single custom function, reducing the total number of instructions down to 4 (bottom-right).

assembly. We tried to work around this issue by writing custom passes that decompose a circuit into logic and non-logic regions, mask out the non-logic regions, apply Yosys' logic synthesis flow, then unmask the non-logic regions. However, Yosys's logic synthesis flow is not restricted to bitwise logic gates and its representation makes it difficult to constrain running logic synthesis on these operations alone. We therefore decided to implement logic synthesis ourselves in the backend Manticore compiler.

Implementation

Custom function synthesis is performed on each partitioned process independently. We start from a process' dependence graph and mask out all non-logic instructions. The result is a *logic subgraph* that contains multiple connected components (unlike the dependence graph which is a single connected component).

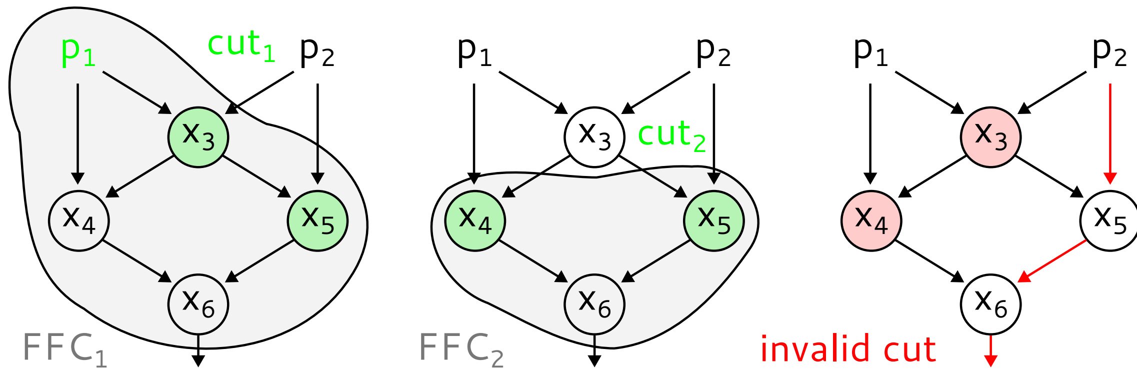


Figure 5.7 – Cuts and fanout-free cones (FFCs) in a logic subgraph. We show a subset of the 3-cuts of vertex x_6. Vertices p_1 and p_2 are the primary inputs. The left and center examples show two valid cuts and their corresponding FFCs. The right example shows an invalid cut as there is a path from x_6 to the primary input p_2 that does not cross the cut.

We exhaustively extract all 4-input *cuts* in each connected component using cut enumeration [33]. Informally, a cut of root vertex v is a set of vertices that separates v from its primary inputs[2] (see Figure 5.7). Cut enumeration is exponential-time as there can be an exponential number of cuts in a general graph. However, we perform cut enumeration on only the connected components of the logic subgraph (not the full dependence graph), so its runtime is tractable (few ms).

The vertices between the root of a cut and the cut's vertices form a *cone*. We keep only fanout-free cones (FFCs), i.e., cones in which the fanouts of every vertex other than the root are in the cone (cone vertices converge to the root) [31]. The fanout property of a cone is important as it allows us to replace the entire cone with a single vertex without having to duplicate internal nodes (i.e., there is no consumer for the internal nodes other than the cone's root vertex). Furthermore, since a FFC contains only logic instructions and has at most 4 inputs, it can be represented as a single 4-input LUT-based custom function.

An important issue is that two cones can have different *representations* while still computing the same 4-input *function*. Consider, for example, the following two cones where `%i` is the ith input of the cone.

- `(%0 | %1 | %2) & %3`
- `(%0 | %2 | %1) & %3`

These cones compute the same underlying function: swapping inputs %1 and %2 in the first cone makes it indistinguishable from the second. Each core supports only 32 custom functions and we would quickly exhaust them if we did not find equivalent cones and implement them with the same custom function.

We consider two cones c_a and c_b as equivalent if a permutation of c_a's inputs produces the same LUT equation in the CFU as c_b. In theory, checking two functions for equivalence is expensive as

[2]The primary inputs of a graph are vertices which have no incoming edge, which are guaranteed to be non-logic instructions in our connected components.

it requires enumerating up to $k!$ input permutations per pair of cores, with k being the maximum number of cone inputs. In practice, the check is not prohibitively expensive as cones can support at most only 4 inputs. As a result we did not try to use more advanced logic equivalence checking techniques.

We cluster cones based on their boolean equivalence before choosing the set of cones to transform into custom functions. Each cluster contains a *representative* cone to which all candidate cones are compared. We initialize a single cluster with the first cone. Each additional cone is then compared against the representative. If the cone is equivalent to the representative, we add the cone to the cluster and store the input permutation that transforms it into the representative. If the cone is not equivalent to the representative of any cluster, then we create a new cluster and mark the cone as its representative.

Finally, we select the best set of FFCs that maximize instruction savings, under the constraints that (1) cones cannot overlap, (2) some FFCs are equivalent and can be implemented by the same function at multiple call sites, and (3) there is a limited number of custom functions available. This is an instance of the set packing problem and is NP-complete [40]. The main difficulty in this problem lies in the non-overlapping constraint, which rules out any greedy algorithm: selecting the best local FFC can rule out many other FFCs which collectively cover more instructions. We use a MILP formulation to obtain an optimal solution as the search space is small (we search for FFCs in the much smaller logic subgraph, not the general dependence graph).

Figure 5.8 illustrates the output of the custom function synthesis optimization on a MIPS processor. Each color represents a different function, one of which is instantiated twice in the graph. Custom functions that contain many constants are larger than others as constants do not count as inputs (they are known at compile time) and are folded into a function's equation (e.g., constants `c680`, `c682`, and `c692` in the pink function).

We describe the MILP formulation next.

Optimization formulation

The inputs of the MILP formulation are (1) the logic subgraph G(V, E), (2) the set C of all FFCs in the process, and (3) the set F of all representative cones. The formulation requires only two variables:

1. A *binary* variable x_c for every cone. Variable $x_c = 1$ if cone c is selected to cover the graph.

$$x_c \in \{0,1\} \quad \forall c \in C$$

2. An auxiliary *binary* variable y_f for every unique function. Variable $y_f = 1$ if function f is called.

$$y_f \in \{0,1\} \quad \forall f \in F$$

We encode the physical constraints of the problem:

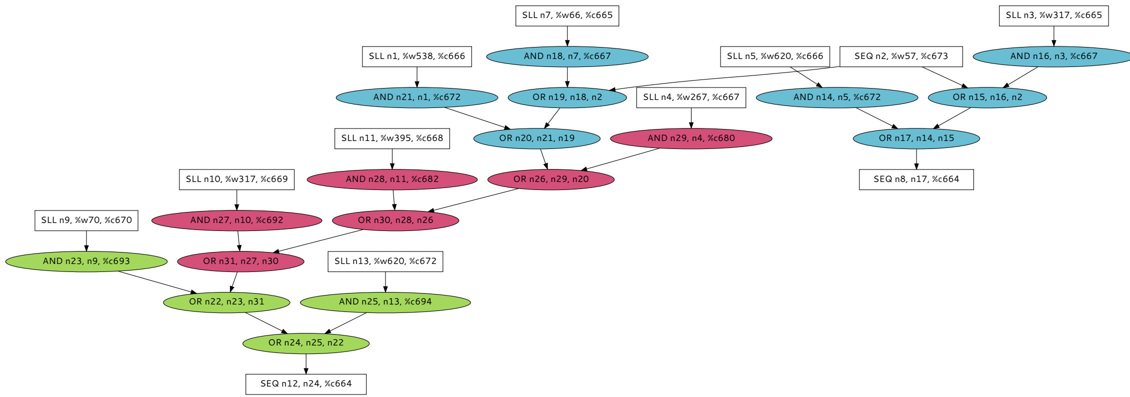


Figure 5.8 – Identifying custom functions in a netlist DAG. We show a small portion of the logic subgraph of a MIPS processor for space constraints. Logic operations are shown with ovals and non-logic operations (which are outside of the logic subgraph) are shown with rectangles. The goal is to maximize instruction savings by covering the most logic operations, while using the fewest custom instructions. Each color represents a different custom function. Two of the functions are called once (pink, green), while one is called twice (blue).

$$
\begin{aligned}
\text{maximize} \quad & \sum_{c \in C} x_c * (|c| - 1) - \sum_{f \in F} y_f \\
\text{subject to} \quad & x_c \in \{0, 1\} && \forall c \in C, \\
& y_f \in \{0, 1\} && \forall f \in F, \\
& \sum_{c \in C | v \in c} x_c \leq 1 && \forall v \in V, \\
& y_f = \bigvee_{c \in C | c \in f} x_c && \forall f \in F, \\
& \sum_{f \in F} y_f \leq MaxFuncts && \forall f \in F
\end{aligned}
$$

Figure 5.9 – Custom function synthesis MILP formulation.

1. Each vertex in the logic subgraph is covered *at most* once. A vertex v is covered if it is covered by a cone c.

$$\sum_{c \in C | v \in c} x_c \leq 1 \quad \forall v \in V$$

2. A custom function is used if any of the cones that it represents are selected to cover the graph. The $\vee$ operator in the constraint below is not linear, but can be linearized following standard transformations [28].

$$y_f = \bigvee_{c \in C | c \in f} x_c \quad \forall f \in F$$

3. The number of custom functions cannot surpass $MaxFuncts$, which is 32 in our design as each core supports at most 32 custom functions.

$$\sum_{f \in F} y_f \leq MaxFuncts \quad \forall f \in F$$

The solver can find an optimal solution in all our benchmarks in less than 1 s.

5.3.4 Scheduling

The scheduler has two main tasks: (1) avoid data hazards in each core, and (2) avoid structural hazards on the shared NoC. It uses a cycle-accurate model of a core's pipeline and the NoC to do so. The model need not contain the details of all system internals. We model only the following details:

- Read-after-write distance in each core (10 cycles);
- Core → switch delay (7 cycles, see Section 4.8.2);
- Switch → switch NoC delay (statically known after placement, see Section 5.3.2).
- Switch → core delay (7 cycles, see Section 4.8.2).

The compiler uses a simple list-scheduling algorithm to avoid data hazards in each core. An instruction is scheduled when its predecessors (in the DAG) are scheduled and executed. If the compiler can choose between scheduling a SEND instruction and an arbitrary other instruction, it prioritizes scheduling the SEND instruction first to overlap communication with computation.

Structural hazards on the shared NoC cannot be handled through list-scheduling alone. The compiler performs an abstract cycle-accurate simulation of one RTL cycle using its model of the core's pipeline and the NoC. A SEND instruction can be issued only when it will not collide with any other messages on its path. We assign a static priority to each core (cores with more instructions have higher priority) to arbitrate between contending SEND instructions from different cores. If we cannot issue an instruction in a scheduling step, the compiler delays it with a NOP instruction. In theory the static priority could lead to starvation as some cores may never be allowed to schedule their SEND instructions, but in practice it does not happen as each core has a bounded number of instructions.

5.3.5 Register Allocation

The compiler maps virtual registers to physical registers with a simple linear-scan register allocator. We optimize redundant register moves by allocating the same machine register to both the current and next values of an RTL register [112]. Manticore's register files have abundant capacity[3], so we do not recycle registers after their lifetime expires for simplicity.

[3]We have 2048 registers for *at most* 4096 instructions, and each instruction generally reads two or more registers, so it is unlikely that we run out of registers.

5.4 Summary

Chapter 3 showed that Manticore's static BSP execution model allows scaling the RTL simulation rate linearly with respect to the parallelism factor if work is *equally distributed* among cores. This is the overarching goal of Manticore's compiler.

The compiler's frontend is built on top of Yosys, which we use to perform behavioral synthesis on a closed[4] input Verilog design. The result of behavioral synthesis is a structural netlist, which we translate into a straight-line sequence of instructions (described in Manticore's ISA) and pass down to the backend.

The compiler's backend heuristically partitions the monolithic assembly sequence into at most N approximately equal processes, where N corresponds to the number of cores available in a Manticore grid. Next, it optimizes each process by fusing long chains of bitwise logic expressions into compact 4-input custom functions using an exact MILP-based optimization formulation. The backend then assigns each logical process to a physical core with a heuristic recursive bi-partitioning algorithm. It continues with a cycle-accurate simulation of execution on Manticore to co-schedule code on all cores so as to (1) resolve data hazards within each core, and (2) avoid structural hazards on the shared NoC. Finally, it uses a linear-scan register allocator to map virtual registers to physical registers in each core.

This concludes our description of Manticore's compiler. Chapter 6 evaluates Manticore's performance and design decisions.

[4]A closed Verilog design exposes only a clock signal (see Chapter 3).

6 Manticore Evaluation

This chapter evaluates Manticore along several dimensions. We first present our baseline software RTL simulator and the benchmarks used in our evaluation. We then report Manticore's performance and break down the impact of the compiler's main optimizations.

6.1 Baseline Simulator

We use Verilator [92] as our baseline RTL simulator. It is a popular open-source full-cycle RTL simulator widely used by academia and industry. Verilator synthesizes Verilog designs into C++ models, which it then interfaces with C++ testbenches. Verilator's primary goal is speed and it is currently faster than other production-grade open-source and commercial simulators [90, 91]. Sources report a ≈ 5–6× speedup [88] compared to VCS (a commercial simulator). Verilator is fast as it takes a synthesis engine's view of circuit simulation, which allows it to perform optimizations that an IEEE Verilog/SystemVerilog-compliant simulator cannot make (e.g., Verilator supports only two-state logic instead of four-state logic). Its generated C++ models are also highly optimized through branch prediction hints, short-circuitable branch conditions, etc.

Verilator supports multithreaded simulation. Figure 6.1 illustrates how it parallelizes RTL simulation. Verilator starts from a netlist DAG similar to Manticore's. Initially each DAG node represents a Verilog statement-level micro-task, i.e., a plain Verilog operator or function call. Verilator increases the granularity of computation by combining adjacent micro-tasks into a small number of *macro-tasks*—the atomic units of work that will run asynchronously across multiple threads—using Sarkar's algorithm [84]. Verilator repeatedly merges the micro-tasks that produce the smallest increase in the critical path of the macro-task. It does so until a heuristic threshold on the critical path length is reached. Each macro-task is then linearlized using the same optimizations Verilator uses for single-thread execution. Finally, macro-tasks are statically assigned to threads using a heuristic concerned with their execution time, and edges between threads become synchronization primitives (user-space spin-locks). At runtime, a macro-task can start its execution when its preceding macro-tasks finish.

Verilator's parallel execution is not BSP since it uses fine-grain synchronization between tasks within a simulated RTL cycle (compute phase). However, in simulating a clock cycle, Verilator

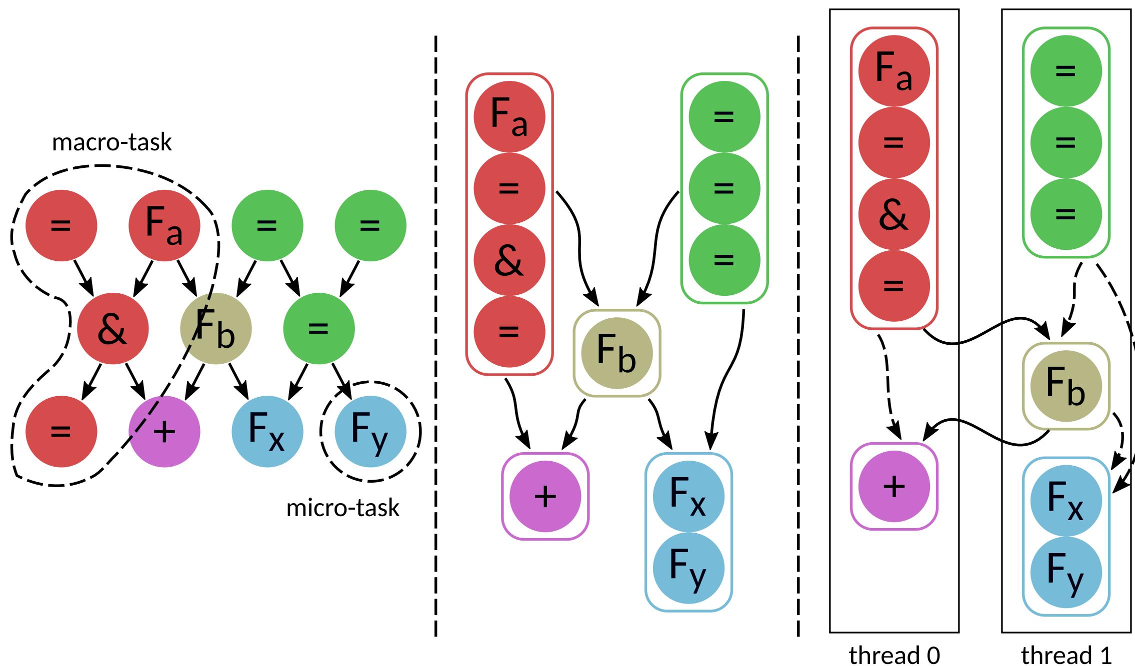


Figure 6.1 – Verilator's multi-threaded compilation flow. The netlist DAG initially contains plain statement-level Verilog micro-tasks (left). Adjacent micro-tasks are grouped together into larger macro-tasks, which are then linearlized and linked with edges that represent synchronization primitives (center). Finally, the macro-tasks are statically assigned to threads and dashed edges marking communication between macro-tasks on the same thread are elided (right).

also uses two synchronization points (final macro-tasks) as a rendezvous for all tasks, similar to the barriers in the model presented in Section 3.4 and in Manticore. Verilator therefore uses *more* synchronization than Manticore.

6.2 Test Environment

We use Verilator v5.006 (February 2023) running on Ubuntu 20.04 LTS. Verilator v5 added support for multiple clock domains and timing. Since Manticore does not yet support these, we disable timing in Verilator to avoid penalizing its performance. We also disable waveform dumps and unnecessary printing to report undiluted performance numbers. We enable all optimizations in both Verilator (i.e., `-O3`) and Manticore (e.g., custom functions).

We evaluate Verilator's performance on a high clock frequency desktop processor (i7-9700K) and two high core count server processors (Xeon 8272CL, EPYC 7V73X). Table 6.1 summarizes the key characteristics of the hardware platforms. Note that the i7-9700K's base clock speed is normally 3.6 GHz, but we overclock it to a base frequency of 4.6 GHz to highlight the importance of clock frequency in RTL simulation. We use `numactl` on Linux to assign Verilator to physical cores on the i7 and Xeon machines (hyperthreading degrades performance as hyperthreads compete for instruction cache space). The EPYC machine does not have hyperthreads.

	Baseline (x86)			Manticore
	Desktop	**Server**	**Server**	
HW	Intel i7-9700K	Intel Xeon 8272CL [107]	AMD EPYC 7V73X [107]	Xilinx Alveo U200
Num. cores	8	32	120	225
Freq. (GHz)	4.6–4.9	2.5–3.4	2.2–3.5	0.475
SRAM (MiB)	14.5	105.5	259.6	18.45
Release date	Q4 2018	Q4 2019	Q1 2022	-

Table 6.1 – Hardware platforms. SRAM capacities for x86 processors correspond to their reported cache size (measured through `lscpu`). SRAM capacity in Manticore is the sum of its cache size and the total capacity of all 225 cores' instruction memories, scratchpad memories, and register files.

6.3 Benchmarks

We evaluate Manticore's performance using nine RTL workloads (the benchmarks are wrapped in simple, assertion-based Verilog test drivers). This benchmark suite covers a wide variety of building blocks: integer, fixed-point, and floating point operations, crypto, pipelined in-order processors and switches, caches, systolic arrays, state machines, and stencil computation:

- **bc** is a bitcoin miner [74].
- **mm** is a 16×16 integer matrix-matrix multiplier. Two matrices are streamed into the array—one being transposed on the fly—and are multiplied with a systolic architecture.
- **cgra** is coarse-grained reconfigurable array (CGRA) of 64 floating-point processing elements. The design is fully latency-insensitive as data flows through the array using small queues. A separate serial "bitstream" is used to configure the grid.
- **vta** is a versatile tensor accelerator [66], i.e., a machine learning accelerator. We use a larger[1] spatial implementation as the default configuration is too small to benefit from hardware acceleration.
- **rv32r** consists of 16 in-order pipelined RISC-V processors communicating over a ring network. The processor implementation is based on that of riscv-mini [58].
- **jpeg** is a pipelined JPEG decoder [45].
- **blur** is a stencil computation accelerator that leverages non-uniform partitioning of data resuse buffers [32].
- **mc** is a Monte-Carlo simulation stock option price evolution predictor with fixed-point arithmetic. Price evolution is computed using fixed-function pipelines based on Monte-Carlo simulation [96].
- **noc** is a 2D 4×4 uni-directional torus network-on-chip with wormhole routing and four virtual channels.

The benchmarks were sized to ensure their state fit in the Manticore on-chip scratchpads, so the compiler can accurately predict performance. We evaluate the global stalling mechanism separately in Section 6.9.

[1] `blockIn` and `blockOut` are set to 64 instead of 16.

		Benchmark →	**vta**	**mc**	**noc**	**mm**	**rv32r**	**cgra**	**bc**	**blur**	**jpeg**	**geomean**	
		# instr. (k) →	169	148	88	74	43	38	20	12	3	**w/ jpeg**	**w/o jpeg**
		# cycles →	1M	1M	1M	1M	1M	1M	2M	5M	1B		
Verilator (kHz)	**i7**	S	41.3	33.9	41.4	43.9	96.6	152.0	599.0	726.7	4246		
		MT	160.2	127.2	80.5	83.0	141.8	146.2	354.4	362.0	700.7		
		×S	3.9	3.8	1.9	1.9	1.5	0.97	0.6	0.5	0.2	***1.19***	***1.48***
	xeon	S	32.4	26.6	37.1	34.7	97.3	136.8	462.7	532.6	3233		
		MT	94.9	68.9	41.5	52.3	73.3	74.3	190.6	186.1	590.6		
		×S	2.9	2.6	1.1	1.5	0.8	0.5	0.4	0.3	0.2	***0.79***	***0.94***
	epyc	S	32.1	29.7	32.4	31.6	109.2	126.0	550.2	430.5	3627		
		MT	146.9	120.8	106.0	95.2	162.7	167.8	370.6	406.9	1239		
		×S	4.6	4.1	3.3	3.0	1.5	1.3	0.7	0.9	0.3	***1.60***	***1.97***
Manticore (kHz)		225-core	278.1	423.0	293.6	567.5	221.0	421.5	1562	1015	214.2		
	i7	×S	6.7	12.5	7.1	12.9	2.3	2.8	2.6	1.4	0.05	***2.75***	***4.54***
		×MT	1.7	3.3	3.6	6.8	1.6	2.9	4.4	2.8	0.31	***2.38***	***3.07***
	xeon	×S	8.6	15.9	7.9	16.3	2.3	3.1	3.4	1.9	0.07	***3.37***	***5.48***
		×MT	2.9	6.1	7.1	10.8	3.0	5.7	8.2	5.5	0.36	***4.16***	***5.66***
	epyc	×S	8.7	14.2	9.1	18.0	2.0	3.3	2.8	2.4	0.06	***3.35***	***5.55***
		×MT	1.9	3.5	2.8	6.0	1.4	2.5	4.2	2.5	0.17	***2.07***	***2.83***

Table 6.2 – Verilator and Manticore simulation performance. **# instr.** is the average number of x86 instructions (in thousands) needed to simulate one RTL cycle. **# cycles** is the number of RTL cycles simulated to measure the simulation rate. For Verilator, the **S** and **MT** rows report the serial and multi-threaded simulation performance in kHz. ×**S** is multithreaded speedup w.r.t. serial. For Manticore, we report simulation rates on a **225-core** configuration, along with the speedup relative to the serial (×**S**) and multithreaded (×**MT**) runs of Verilator. We color entries that show speedups in green and entries that show slowdowns in red. We report geomean speedups with and without the **jpeg** benchmark.

Section 6.4 compares the simulation rate of Verilator against that of a 475 MHz, 225-core Manticore processor. Section 6.5 continues by comparing Verilator and Manticore's performance scaling trends as we increase the number of available cores in each platform. Section 6.6 evaluates the compiler's contribution to Manticore's performance in more detail. Section 6.7 continues with an evaluation of Manticore's compile time. Section 6.8 studies Manticore's cost if run in a cloud environment. Finally, Section 6.9 evaluates Manticore's global stalling mechanism and its performance penalty.

6.4 Performance Comparison

We run each benchmark for millions to billions of cycles to capture steady-state performance. Table 6.2 reports the simulation rates and speedups achieved by Manticore and Verilator. Figure 6.2 plots the simulation rates reported in Table 6.2 for simpler inspection.

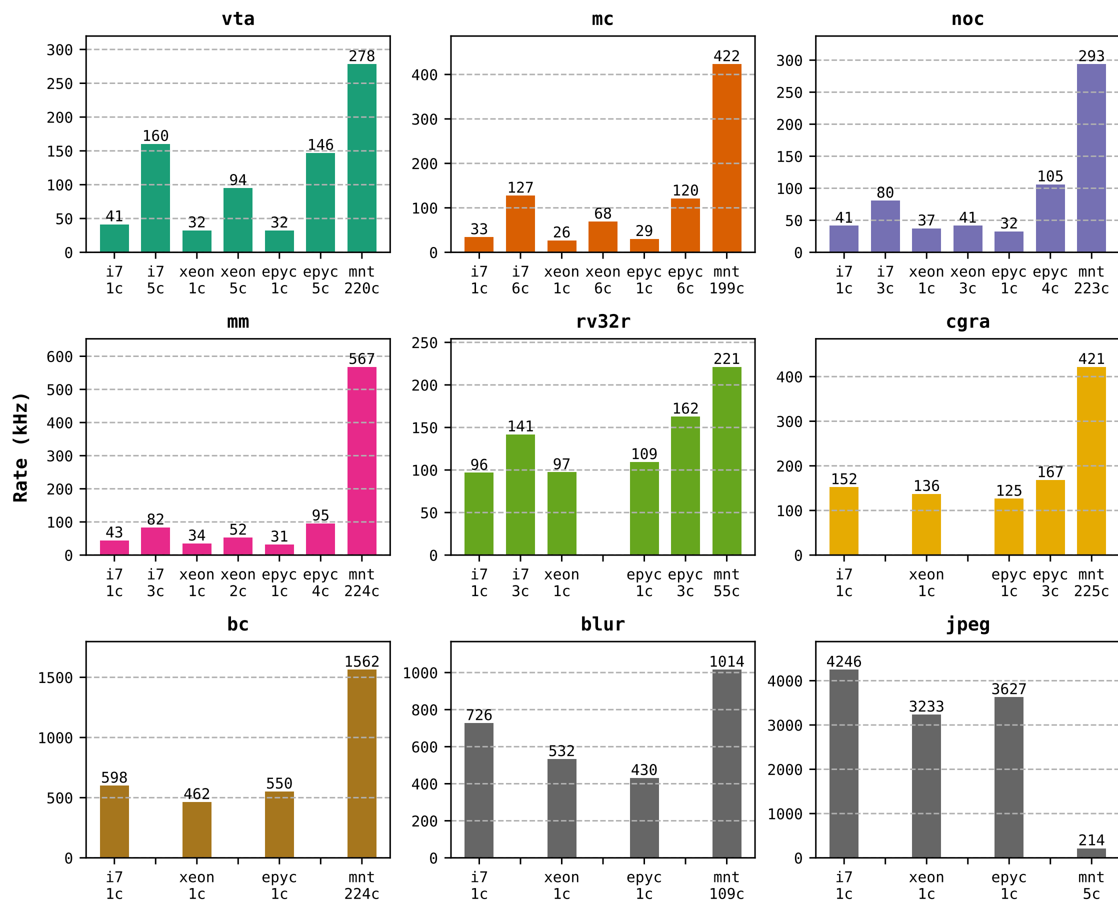


Figure 6.2 – Verilator and Manticore simulation rate. We report the serial and multithreaded simulation rate for the desktop (i7) and server (xeon, epyc) processors, and for a 225-core Manticore (mnt). The numbers underneath the x-axis denote the number of threads/cores used in each platform. For multithreaded Verilator we report the number of threads used to obtain the *best* performance (by sweeping the thread count). We omit multithreaded execution entries when multithreading degrades Verilator's performance w.r.t. serial execution. Manticore's compiler automatically finds a program partitioning which attempts to maximize the amount of parallelism used (i.e., we need not sweep the core count).

6.4.1 Verilator

The top half of Table 6.2 and the left columns in Figure 6.2 report Verilator's serial (**S**) and multithreaded (**MT**) simulation rates separately for each hardware platform. We select the best multithreaded simulation rate for each benchmark.

Multithreaded Verilator improves performance w.r.t. serial execution by up to 3.9× and 4.6× on desktop and server processors, respectively. Multithreading could not improve performance on the smaller benchmarks (**bc**, **blur**, and **jpeg**). All processors reach their scalability limit with fewer than 8 threads. Given the number of instructions in each step of the benchmarks, these results agree with the abstract simulation model discussed in Section 3.4.

6.4.2 Manticore

Our largest Manticore configuration is a 16×16 grid running at 450 MHz. While Manticore benefits from increased parallelism, most of our medium-sized benchmarks do not scale past 225 cores, and so benefit more from a higher-clocked 15×15 configuration clocked at 475 MHz. The bottom half of Table 6.2 and the rightmost column in Figure 6.2 therefore report the simulation rates on a 475 MHz, 225-core Manticore. The table also reports speedups relative to Verilator's serial (×**S**) and multithreaded (×**MT**) performance. Manticore exploits parallelism to speed up simulation performance and is faster than Verilator in 8 of the 9 benchmarks.

Manticore obtains a large 2.5–6× speedup over Verilator's best performance on x86 processors in benchmarks where the compiler can find enough parallelism to make use of most of Manticore's available cores (**mc**, **noc**, **mm**, **cgra**, and **bc** use close to 200 of Manticore's 225 cores). Note that **cgra** and **bc** are ≈2–7× smaller than the other benchmarks, but still exhibit a much higher simulation rate than Verilator, demonstrating the benefit of exploiting parallelism to speed up RTL simulation even in smaller designs. Our largest benchmark, **vta** is a slight outlier here: while the compiler can find enough parallelism to use 220 cores, these cores suffer from imbalanced workloads (see Section 6.5) and therefore limit Manticore's speedup against Verilator's best performance on x86 to at most 1.7×.

Manticore obtains a smaller 1.4× speedup over Verilator's best performance in benchmarks where the compiler cannot find enough parallelism to use most of Manticore's cores (**rv32r** and **blur** use only 55 and 109 of Manticore's 225 cores, respectively).

The **jpeg** benchmark is the only one where Verilator outperforms Manticore. This benchmark has the highest simulation rate in Verilator and the lowest in Manticore. The **jpeg** benchmark contains sizeable sequential data dependencies that cannot be parallelized (a long chain of Verilog `if-else` statements for the Huffman table lookup). Manticore's slow sequential performance hurts us on this serial benchmark[2]. Parallelism improves **jpeg**'s single-core performance by only ≈17%. This marginal improvement cannot compensate for the single-core disparity between Manticore and x86.

6.5 Scaling Trends

6.5.1 Performance

Figure 6.3 plots each machine's speedup scaling w.r.t. serial execution. Manticore's speedup numbers here are predicted by the compiler instead of actual execution since it can accurately count cycles in the absence of off-chip memory accesses. The compiler reports a *virtual critical-path length (VCPL)*, the total number of instructions (including `NOP`s) in the slowest core. VCPL is the number of Manticore machine cycles (i.e., FPGA cycles) required to execute one RTL cycle. We consider the single-core VCPL as the baseline to demonstrate Manticore's scalability. Note, however, that single-core execution on our prototype is—for most benchmarks—impossible since

[2]Note that the jpeg *algorithm* is parallelizable, but the hardware *implementation* we use as a benchmark [45] is serial.

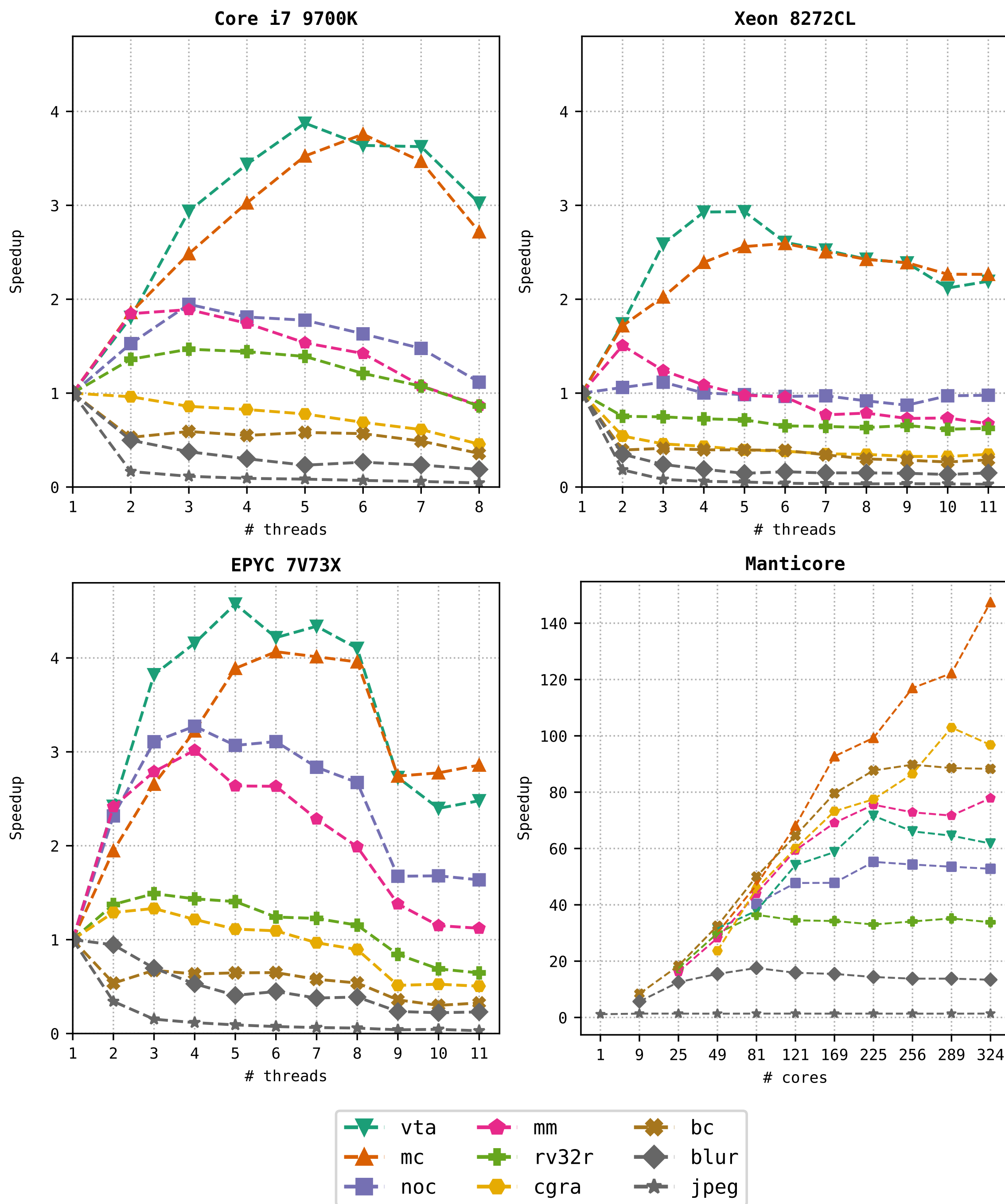


Figure 6.3 – Verilator and Manticore simulation scaling. The desktop (upper-left) and server (upper-right, bottom-left) numbers are obtained from Verilator. We use the same y-axis for the general-purpose processors to highlight the different in each processor's performance scaling. The Manticore (bottom-right) numbers are obtained from Manticore's compiler. The general-purpose processors all exhibit the same performance trend and speedup eventually decreases as more threads are used. By contrast, Manticore's performance—in general—monotonically increases as we use more cores and plateaus when the compiler can no longer find parallelism.

there is not enough space in a single core's instruction memory (only **jpeg** can fit entirely in a single core).

The figure best illustrates the stark constrast between the shared-memory processors' scalability and Manticore's: the x86 machines can only marginally increase performance before witnessing a decrease in performance, whereas Manticore continues to improve performance as the number of cores increases to 200–300. In particular, notice the order of magnitude difference in scale between the y-axes of the different machines. Note, however, that Manticore's performance gain through parallelism must be weighed against the single-core/thread performance disparity between a Manticore and an x86. To match the serial performance of a 4.6 GHz to 4.9 GHz desktop processor, Manticore must overcome a 10× lower clock speed. Furthermore, general-purpose x86 processors can execute 1–2.5 instructions-per-cycle (IPC). Manticore's simple processors execute a single instruction per cycle, have a narrower datapath (16 bits vs. 32 bits), and support only simple instructions. Manticore can match the desktop processor's serial performance only if it can achieve a performance improvement of at least 10–25× by employing parallelism effectively. In other words, a large fraction of the gain goes into making up for the loss in single-core performance. However, the measurements demonstrate that, with appropriate architectural and compiler support for fine-grain parallelism, we can reach simulation speeds that are *unattainable* on a general-purpose architecture.

Finally, Manticore is not immune to Amdahl's law. If there is insufficient parallelism in the workload, then Manticore's scaling plateaus. Depending on the RTL design, this may happen early (**jpeg**) or late (**mc**).

6.5.2 Workload Distribution

Achieving good parallel speedup hinges on (1) using as many of Manticore's available cores as possible, and (2) ensuring cores have relatively balanced workloads. The partitioner's main goal is to produce balanced program partitions, while simultaneously attempting to minimize communication between partitions.

Figure 6.4 analyzes the workload distribution across cores as we increase Manticore's total core count using a violin plot (a box plot which also shows the density of values in each box). We model cores as having infinite instruction/scratchpad capacity so that the benchmarks fit in even small Manticore hardware configurations. For each hardware configuration we normalize cores' program lengths by their average to compare program length deviations in all benchmarks using the same scale on the y-axis. We consider only the "compute" portion of each core's workload as it is the metric that the partitioner attempts to balance. The partitioner runs *before* placement and scheduling, and so it uses only the number of instructions (including `SEND`s) in each partition as the primary metric while balancing workloads. However, analyzing workload distributions makes sense only *after* scheduling when `NOP`s have been inserted to handle data dependencies within each core and structural hazards on the shared NoC. We therefore count instructions in each core only up to the point when the core has finished its final `SEND` instruction (i.e., we

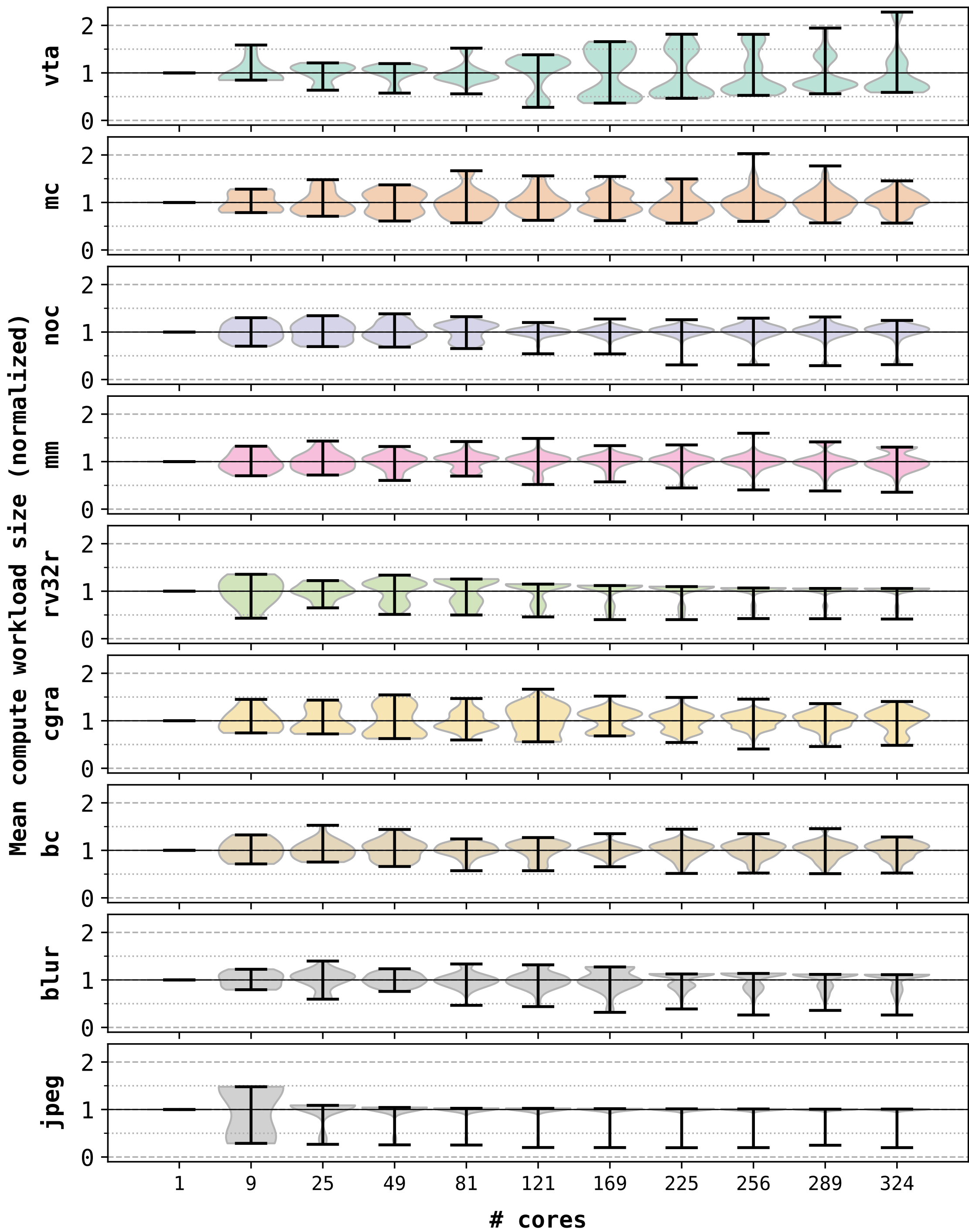


Figure 6.4 – Workload imbalance between cores. The x-axis denotes the number of cores in each Manticore configuration. The y-axis shows a violin plot (box plot with density) of program lengths across all cores. We normalize program lengths by their average so they are all comparable on the same scale (i.e., for each benchmark the average program length is represented by the solid black line at y-value 1). A y-value of 0 denotes program partitions that have 0 instructions. A y-value of 2 denotes a program partition whose length is equal to 2× that of the average.

exclude all NOPs inserted afterwards when the core is exclusively waiting for incoming messages to be received).

As expected, workload distributions are not perfectly balanced: program lengths for different cores in the same benchmark vary between 0.25–2.2× the average program length. In general, Manticore can achieve good speedups for benchmarks that use most of its cores, and where most cores' program lengths are centered around the average. This accords with the large gap in simulation rate between Manticore and the x86 processors seen in Figure 6.2 for **mc**, **noc**, **mm**, **cgra**, and **bc**.

As we saw in Figure 6.2, the **vta** benchmark does not achieve as high of a speedup on Manticore compared to an x86 despite using most of Manticore's available cores. The top-most sub-plot in Figure 6.4 shows that the workload distribution among cores in **vta** is highly unbalanced: most cores have program lengths around 0.5× or over 2.2× the average program length.

The **rv32r** and **blur** benchmarks have similar workload distributions: most cores are centered around the average program length, with the exception of a small fraction of cores which have much shorter programs. These benchmarks, however, do not have enough parallelism to use most of Manticore's cores, and so achieve a smaller overall speedup (see Figure 6.3).

Finally, **jpeg** does not have enough parallelism to use more than 5–6 cores, and so is not a suitable benchmark for execution on Manticore.

6.5.3 Instruction Duplication Overheads

Recall that Manticore's partitioner duplicates all shared intermediate vertices while doing a bottom-up traversal of the netlist DAG starting from all *next* register values. These duplicated instructions allow partitions to run in parallel independently at the expense of additional compute. The partitioner accounts for this duplication and attempts to eliminate it when merging two partitions, but duplicated instructions nevertheless do contribute to each core's workload.

Figure 6.5 reports the proportion of instructions in all cores that are duplicated instances of an instruction in the unpartitioned netlist DAG. We report the number of duplicated instructions during partitioning as this information is lost in the compiler afterwards. The overheads reported therefore do not account for NOPs that are inserted by the scheduler, and so the proportions will be lower in the final program.

Higher core counts require more duplication to make partitions independent and fit within Manticore's BSP model of execution. The proportion of duplicated instructions in the whole program increases with maximum core counts. Most benchmarks can benefit from parallelism by replicating fewer than 20% of the unpartitioned netlist DAG's instructions with 324 cores. Two outliers are **noc** and **blur**, where duplicated instructions account for 45% of the total program size. The **blur** benchmark is very small (our 2$^{\text{nd}}$ smallest benchmark), and so instruction duplication makes a larger impact on total program size. The **noc** is a large benchmark and has many

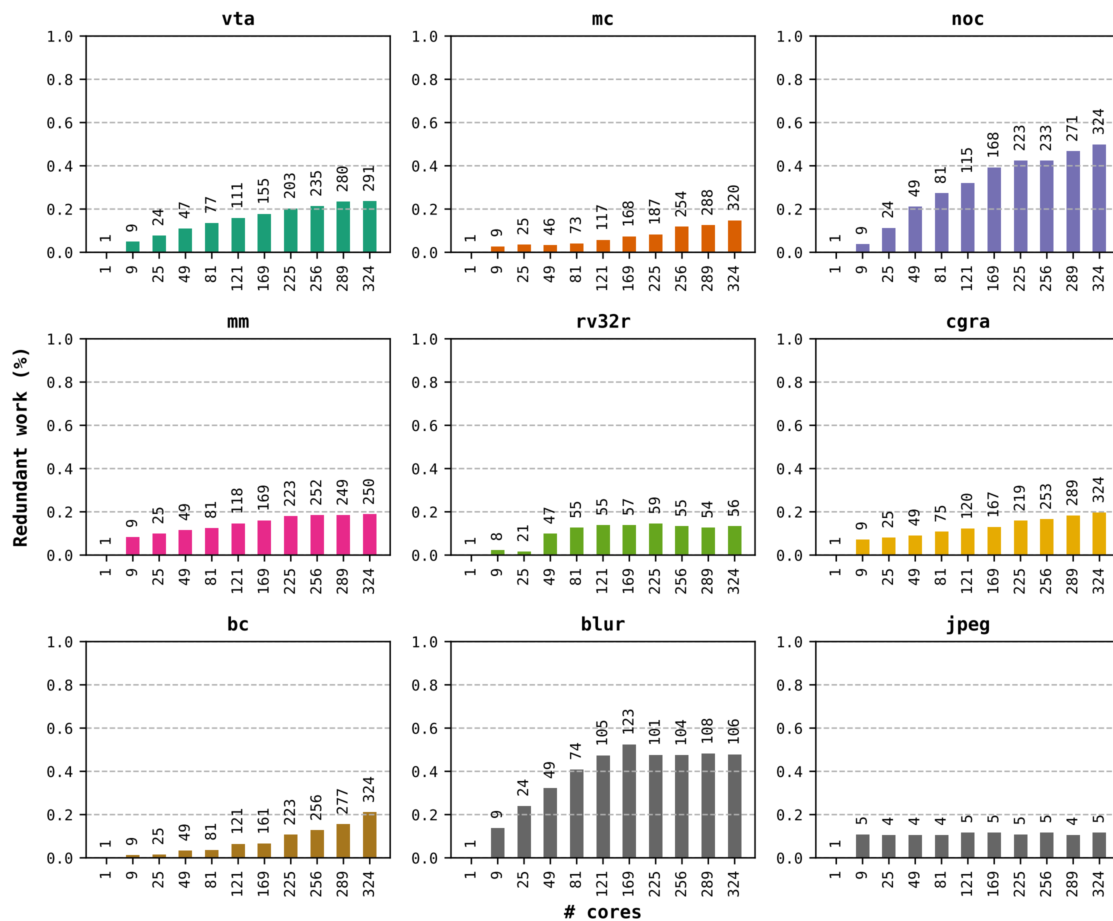


Figure 6.5 – Instruction duplication overhead when scaling Manticore designs. The x-axis represents the total number of cores in a given square Manticore configuration. The y-axis quantifies the proportion of instructions, across all program partitions, that are duplicated instances of an instruction in the unpartitioned netlist DAG. The numbers above each bar denote the number of cores that Manticore's compiler could effectively use in each configuration. Numbers here are reported by the compiler's partitioner, and so do not account for NOPs that are inserted by the scheduler in later stages of the compilation flow.

small combinational operations that fan out to multiple registers, and so these combinational operations end up being duplicated much more than in other benchmarks.

6.5.4 Where Is Time Spent?

Manticore can suffer from two types of compiler-inserted stalls (NOPs): compute stalls and network stalls. Manticore's compiler schedules SEND instructions that are ready (operand is available) as early as possible to avoid congesting the shared NoC. If no SEND instruction can be scheduled, then the compiler schedules any other ready instruction.

A compute stall occurs when there is no SEND instruction available to schedule (operand is not ready or the NoC is busy) and when no other instruction can be scheduled (operands are not

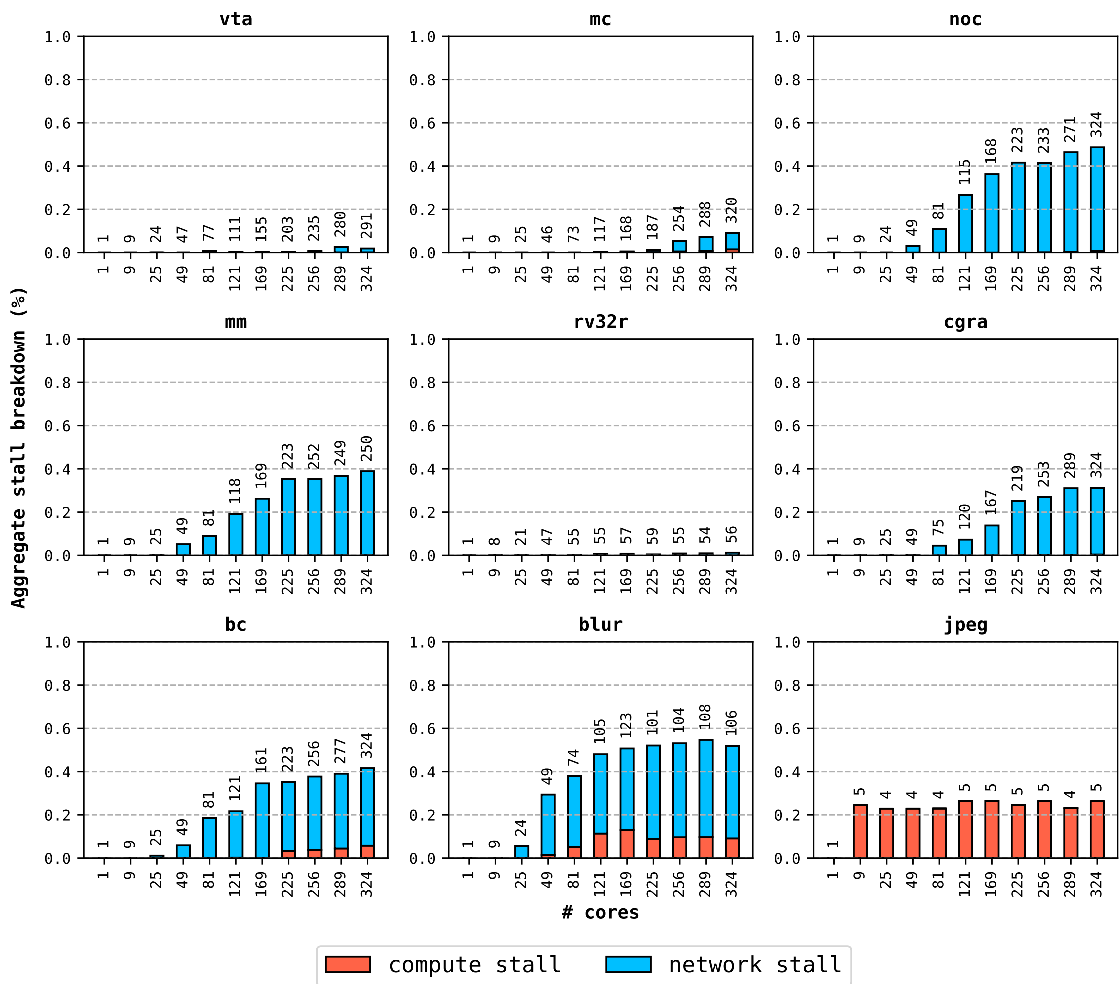


Figure 6.6 – Manticore aggregate stall breakdown. The x-axis denotes the number of cores available in each Manticore configuration, and the y-axis denotes the aggregate proportion of NOPs of each category across all cores. The labels above each bar represents the actual number of cores used by the compiler.

ready). A network stall occurs when a SEND instruction is ready to be scheduled, but the network is busy and there is no other ready instruction to schedule. Since network stalls can occur only when no other instruction could be scheduled, a high proportion of network stalls would signal an architectural bottleneck in Manticore's NoC. By contrast, compute stalls represent bottlenecks in the design of Manticore's cores as a better core could shorten read-after-write delays and make more downstream instructions available for scheduling if the network is busy.

Figure 6.6 breaks down the cause of aggregate compiler-inserted stalls as we scale Manticore designs. Both compute and network stalls generally increase as more cores are used, with network stalls far surpassing compute stalls in proportion to the total number of instructions.

The **vta**, **mc**, and **rv32r** benchmarks exhibit at most ≈ 8% aggregate compute or network stalls w.r.t. the total number of instructions: there is enough work to keep cores busy (compute) when

the network is busy, and so SEND instructions can be delayed without stalling the cores until the network is free.

The **noc**, **mm**, **cgra**, **bc**, and **blur** benchmarks exhibit at most 5% compute stalls, but suffer from $\approx 40\%$ network stalls: the cores have generally balanced workloads (see Figure 6.4), but need to communicate a lot, and so end up stalling for access to the network.

The **jpeg** benchmark is an outlier and suffers from only compute stalls. This benchmark has a long chain of dependencies which results in an excessively long schedule due to Manticore's 10-cycle read-after-write penalty, and so compute stalls account for over 20% of the total runtime. However, the **jpeg** benchmark is very small and doesn't have much parallelism. Larger benchmarks rarely exhibit compute stalls, which is in accord with our hypothesis that there is ample parallelism in RTL simulation to fill in empty pipeline slots in each core's instruction sequence.

Evidently, Manticore's NoC design is a bottleneck and does not have enough bandwidth. Future designs should focus on improving this point.

6.6 Compiler Optimizations

This section evaluates the compiler's contribution to performance.

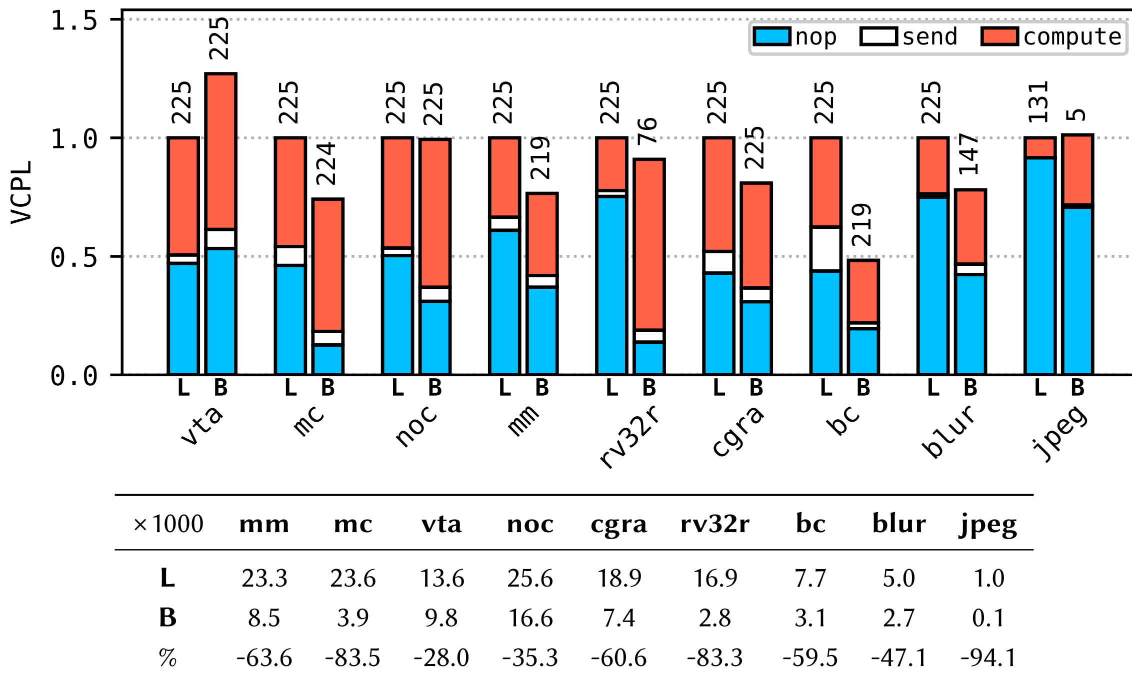
6.6.1 Communication-Aware Partitioning

The balanced partitioning algorithm (B) described in Section 5.3.1 merges the split processes while keeping communication costs low. As a baseline, we compare it against communication-oblivious, longest processing-time first partitioning (L) to observe the benefits of modeling communication. Both algorithms are heuristic and use the same cost estimation method but differ in their merge strategy. Furthermore, both algorithms are oblivious to the effects of instruction scheduling (after partitioning) as neither accounts for the NOPs inserted to avoid data hazards and NoC contention.

Figure 6.7 compares the two approaches for a 15×15 Manticore grid, with VPCL normalized to that of L. We divided the VCPL into the fraction of cycles in the straggler spent computing (compute), sending messages (send), or doing nothing (NOP). Modeling communication is beneficial as B significantly reduces the overall number of SENDs (see table in Figure 6.7), reduces the number of NOPs in the straggler, and generally outperforms (except for **vta**) the communication-oblivious algorithm (L) while using fewer cores. The quality of partitioning significantly affects performance, as evident with **bc** and **mm**.

6.6.2 Custom Instructions

We initially proposed custom instructions to compensate for the lack of instruction-level parallelism in Manticore's simple processors by exploiting bit-level parallelism seemingly abundant in RTL. This bit-level parallelism is captured by FPGA place-and-route tools and compresses a circuit's critical path through high radix LUTs. We sought to create something similar in



×1000	mm	mc	vta	noc	cgra	rv32r	bc	blur	jpeg
L	23.3	23.6	13.6	25.6	18.9	16.9	7.7	5.0	1.0
B	8.5	3.9	9.8	16.6	7.4	2.8	3.1	2.7	0.1
%	-63.6	-83.5	-28.0	-35.3	-60.6	-83.3	-59.5	-47.1	-94.1

Figure 6.7 – Communication-oblivious vs. communication-aware partitioning. **L** is a communication-oblivious, longest processing-time first partitioning algorithm and **B** is our communication-aware partitioning algorithm from Section 5.3.1. We report performance for a 15 × 15 Manticore grid where we normalize VCPL to the VCPL of **L**. The numbers above each bar are the number of cores used by each algorithm. The table reports the number of SEND instructions (in thousands) produced by longest processing-time first partitioning (**L**) and balanced partitioning (**B**).

Manticore's FPGA prototype, though we definitely lose the power of an FPGA's routing circuitry as the CFU supports only bit-parallel LUT-based computation.

Figure 6.8 shows the VCPL of each benchmark normalized to the VCPL without custom instructions. The VCPL is divided among custom instructions, NOPs, and other instructions. The numbers above each bar show the reduction in the total number of instructions over *all* cores (excluding NOPs). This reduction is of 2.9–17.8%, yet the VCPL reduction (i.e., end-to-end reduction) is less than 10% for all benchmarks. In effect, custom instructions reduce the *total* instruction count, but may not reduce the path length of the straggler (e.g., in **mm**). In summary, custom functions consistently improve VCPL, but cannot help improve performance if the critical path consists of non-logic instructions. Their benefit comes with a small cost of one BRAM and a few hundred LUTs per core. Eliminating the custom instructions would not enable larger Manticore grids since the URAMs are the limiting resource.

The compiler extracts custom functions *after* partitioning. We tried moving custom function synthesis *before* partitioning to provide the partitioner with better estimates of execution time. We set the maximum number of functions to be unlimited and revert the excessive ones after partitioning to fit within the hardware limit. However, moving custom function extraction to before partitioning did not make any difference in final VCPL.

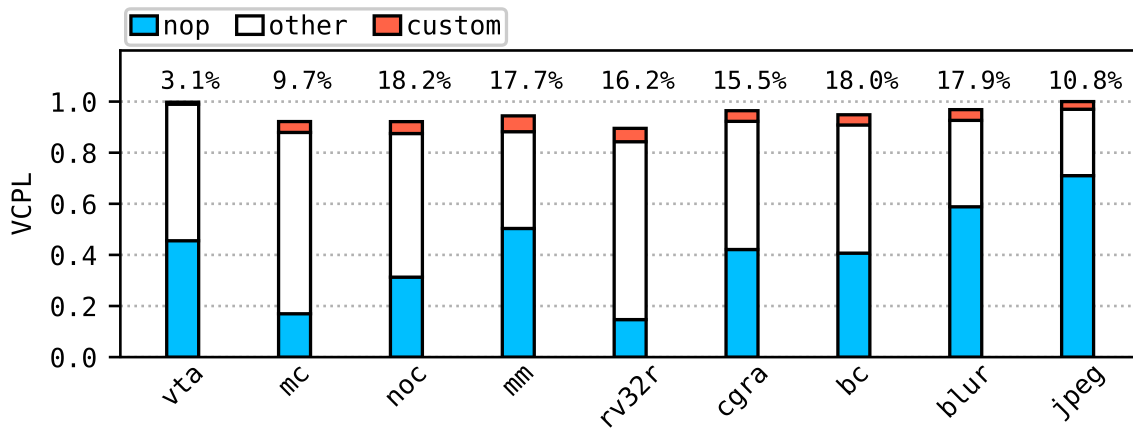


Figure 6.8 – Custom functions' contribution to performance. The VCPL is divided into three instruction types and normalized to not using custom functions. The numbers above each bar represent the reduction in non-NOP instructions over all cores. Custom functions reduce the *total* instruction count by up to 17.8%, but do not specifically target the critical path of the straggler, so end-to-end improvements are less than 10%.

6.6.3 Placement

Our hypothesis was that reducing aggregate traffic on NoC links would result in a shorter global schedule as there would be less contention for its shared links during scheduling. We showed that finding an optimal solution was intractable, so we developed a heuristic instead where we recursively bi-partition processes to assign them to cores (see Section 5.3.2).

However, our experiments showed no *consistent* improvement in VCPL between (1) the bi-partitioning placement algorithm, and (2) simply assigning processes to cores in the order we receive them from the partitioner. The bi-partitioning heuristic assigns processes to cores in a coarse way and cannot see how their placement affects NoC contention any better than a "random" placement.

Placement algorithms can help reduce only communication-related stalls, and so we performed a control experiment to better understand the effect of program placement on performance. We augment the compiler with an ideal, fully-connected "atomic" NoC model. In this model, connections between cores form a complete graph (direct connections between cores) and so NoC links are never busy and do not stall. However, we do model a receiving core's single NoC ingress port to observe port contention. Figure 6.9 reports the normalized reduction in VCPL of this atomic NoC model (**A**) against the VCPL of Manticore's torus NoC (**T**).

Some benchmarks see no performance benefit with an ideal NoC (**mc**, **rv32r**, **jpeg**). In these benchmarks, the straggler is either compute-bound (i.e., the straggler has a long chain of non-SEND instructions), or is port-limited at its NoC interface. Other benchmarks experience a 10–20% reduction in VCPL (**vta**, **noc**, **mm**, **cgra**), and **bc** sees the most benefit with a ≈ 25% reduction. These benchmarks all have a large amount of overlapping communication—generally at the same time—and are heavily NoC stalled with Manticore's torus NoC.

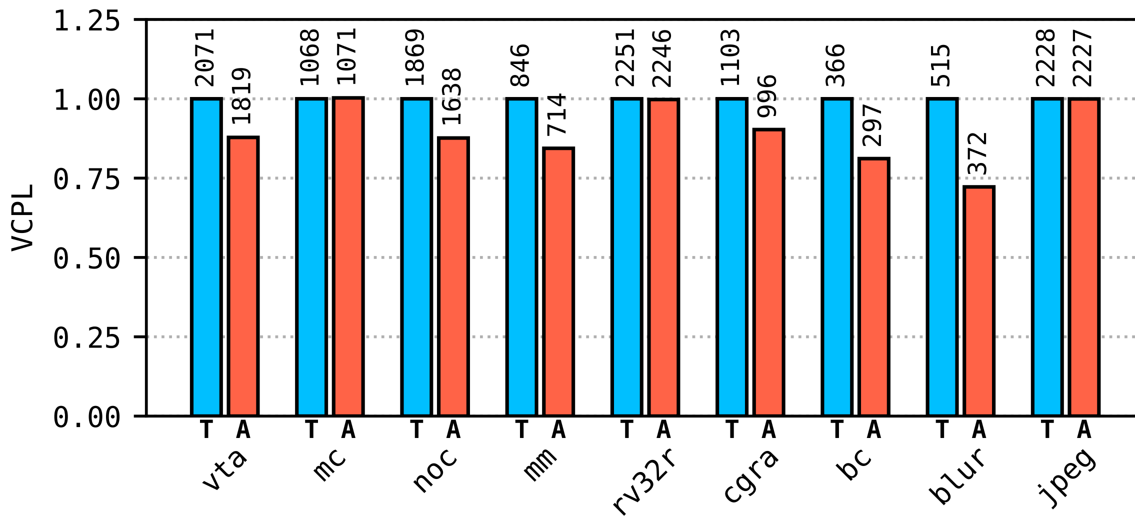


Figure 6.9 – Ideal NoC performance. **T** is Manticore's torus NoC, and **A** is an ideal "atomic" fully-connected NoC with infinite bandwidth (i.e., NoC links are never busy, but target cores still have only one write port at their NoC interface). We report the achieved VCPL of the atomic NoC normalized to that of Manticore's torus NoC. The numbers above each bar represent the absolute VCPL of each benchmark.

In summary placement contributes to overall performance, but its contribution to end-to-end performance is dwarfed by those of partitioning and scheduling due to Manticore's simple NoC and poor port bandwidth at receiving cores. A more sophisticated NoC would be needed to reap the performance benefits of placement.

6.6.4 Register Usage

Recall that the compiler does not recycle register names past their lifetime for simplicity as we have abundant register capacity. In practice this results in high register utilization (up to ≈ 80% in some designs), but never causes any spilling. We analyze register lifetimes during execution to obtain better insights about register capacity.

Figure 6.10 reports the number of live registers at each clock cycle for all benchmarks on our best Manticore configuration (15 × 15). The x-axis represents the number of clock cycles needed to simulate one RTL cycle. The y-axes depict the global number of live registers in the entire Manticore grid (left) and the maximum number of live registers in all cores (right). We observe that the number of live registers at a given instant can reach up to ≈ 30% of a core's register file capacity (rv32r). However, aggregate register file capacity in the entire Manticore grid is high, so we do not observe more than ≈ 8% utilization (noc).

Our goal with using a large register file in the Manticore architecture was to not have to load and store values from a data memory at each simulated RTL cycle so Manticore's limited instruction space is reserved for computation. The analysis above shows that the register file in one core easily has the capacity to contain the working set of 2–3 cores, but our current register file design does not have enough port bandwidth to support this.

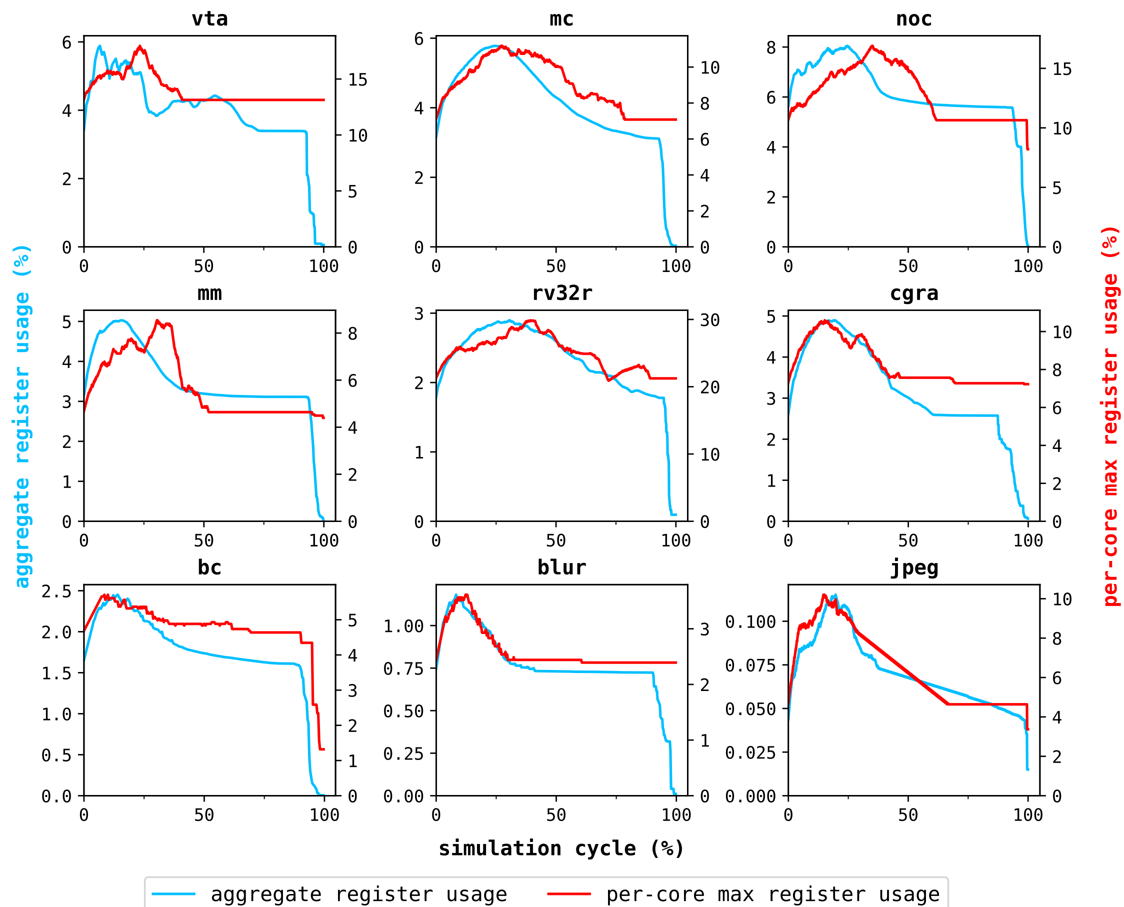


Figure 6.10 – Register file lifetime analysis on a 15 × 15 Manticore grid. The x-axis represents the number of clock cycles needed to simulate one RTL cycle (normalized). The left y-axis plots the global number of live registers in the entire Manticore grid at each clock cycle, whereas the right y-axis plots the maximum number of live registers in all cores. Manticore's large register files have abundant capacity as at most ≈ 30% of the available registers in *one* core are needed to implement our most register-hungry benchmark (**rv32r**). Similarly, global register file capacity is highly under-utilized, with at most ≈ 8% use (**noc**).

6.7 Compile Time

Table 6.3 reports end-to-end compile times for Manticore, single/multithreaded Verilator compilation. The Manticore compiler is a prototype built in Scala for robustness. Its compile times can be several minutes (max. 16 min). By contrast, Verilator compilations usually take less than a minute. Figure 6.11 breaks down the Manticore compiler's execution time into its various components.

Despite its compilation time, Manticore offers a software development-like experience for longer simulations. For example, simulating 10B cycles of the **vta** takes about 10 h on Manticore and 17 h on the i7. In many cases, the extra compilation times are more than compensated by the increased speed.

Benchmark	$\lvert E\rvert$	$\lvert V\rvert$	LoC	Compile Time		
				Manticore	Verilator	Verilator MT
vta	56142	7037	190818	15m29s	2m33	26s
mc	52330	9182	30353	12m57s	1m13s	16s
noc	114364	6927	39363	15m14s	3m23s	36s
mm	89102	6659	64963	8m38s	7m5s	2m55s
rv32r	60430	4497	31761	5m57s	1m56s	29s
cgra	57532	4615	104498	7m48s	2m15s	37s
bc	8135	4630	276	2m23s	40s	27s
blur	9649	751	3869	42s	22s	15s
jpeg	1005	131	6542	16s	7s	3s

Table 6.3 – Compile times across all platforms. We report compile times for Manticore, Verilator with single-threaded compilation, and Verilator with multithreaded compilation. We sort entries in descending order of Manticore's compile time. $\lvert E\rvert$ and $\lvert V\rvert$ respectively denote the number of edges and nodes in the graph obtained by splitting each benchmark into a maximal set of independent processes (see Section 5.3.1). **LoC** denotes the Verilog lines of code for each benchmark.

Current Manticore compile times are longer than a conventional compiler. This is not an inherent limitation but a byproduct of building a research compiler that allows us to explore alternatives rather than a fast compiler. Nevertheless, the current compiler offers a faster time-to-result than parallel software simulation for even hour-long simulations. Most of the compilation time is spent in partitioning the netlist DAG. Partitioning is necessary for parallel RTL simulation, irrespective of the target hardware (e.g., x86 or Manticore). The higher degree of parallelism in Manticore makes partitioning more expensive. We could close the gap between Manticore's and Verilator's compile times with some engineering effort. In addition, algorithms from research on high-quality, low-complexity partitioning could help in this step [17, 22, 37, 59, 83, 115].

6.8 Cost Analysis

For completeness, we provide a brief cost analysis using prices from Microsoft Azure. We estimate the cost of running a few billion simulation cycles in the cloud. Table 6.4 shows the Azure instances used in this analysis. We use the **D2 v4** instance with two virtual CPUs (vCPU) for serial simulation. For multithreaded simulation with Verilator, we use the **D16 v4** instance with sixteen vCPUs. Furthermore, we also consider the **HB120rs v3** instance as it lists RTL simulation as a use case. Renting individual cores on this instance is impossible; therefore, we consider this instance type for only parallel simulation. Unfortunately, renting an FPGA with a single vCPU in Azure is also impossible. The smallest instance is the **NP10s** with one Alveo U250 FPGA board and ten vCPUs, which makes the FPGA instance disproportionally expensive since we also pay for the unused cores.

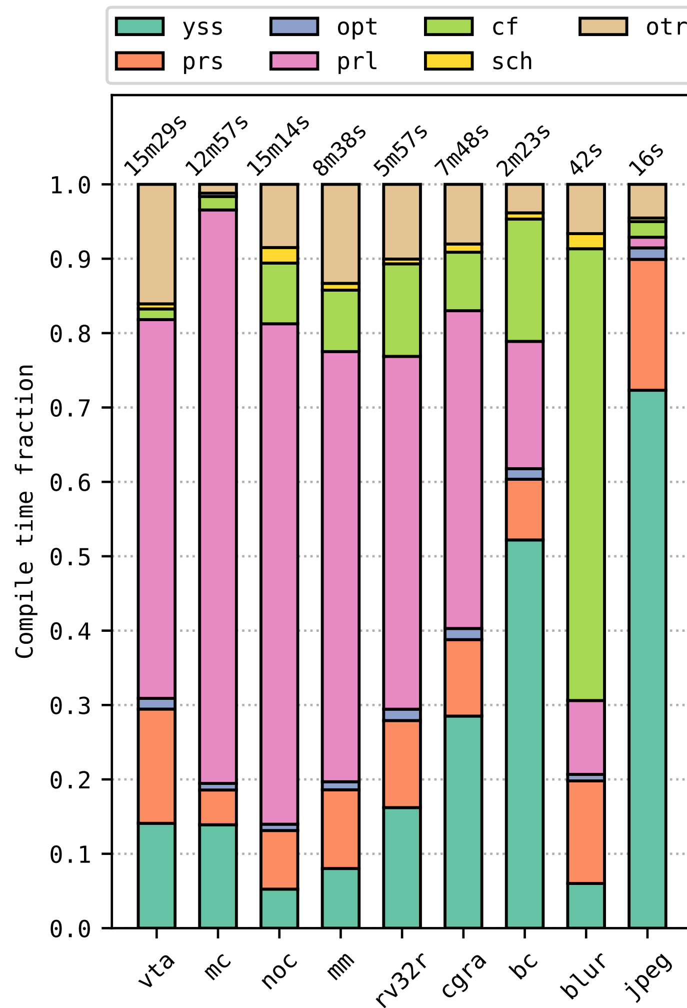


Figure 6.11 – Breakdown of compilation time. Yosys (yss), assembly parsing (prs), basic optimizations (opt), parallelization (prl), custom function extraction (cf), scheduling (sch), others (otr).

Instance	Resources	$/hour	Simulation
D2 v3	Xeon 8272CL (2× vCPU)	0.115	serial
D16 v4	Xeon 8272CL (16× vCPU)	0.92	multithreaded
HB120rs v3	EPYC 7V73X (120× vCPU)	4.68	multithreaded
NP10s	Xeon 8171M (10× vCPU) + Alveo U250	2.145	Manticore

Table 6.4 – Hourly cost of Microsoft Azure instances. Prices reported for February 2023 [7].

All simulations finish in less than an hour for runs shorter than one billion RTL cycles (not shown). With hourly pricing and Verilator's sublinear speedup, serial execution would be the least expensive, followed by multithreaded D16 and Manticore, and finally the HB-series. However, the cost differences are small (few dollars at most). More realistically, we consider 1 and 10-billion cycle multiple-hour simulations by estimating the execution time using the simulation rates from

			vta	mc	noc	mm	rv32r	cgra	bc	blur	jpeg
1B	D2	h	**8.58**	**10.45**	7.48	8.00	2.85	2.03	0.60	0.52	0.09
		$	[1.04]	[1.27]	[0.92]	[0.92]	[0.35]	[0.35]	[0.12]	[0.12]	[0.12]
	D16	h	2.93	4.03	6.70	5.31	2.85	2.03	0.60	0.52	0.09
		$	2.76	4.60	6.44	5.52	2.76	2.76	0.92	0.92	0.92
	HB	h	1.89	2.30	2.62	2.92	1.71	1.66	0.50	0.52	0.08
		$	9.36	14.04	14.04	14.04	9.36	9.36	4.68	4.68	4.68
	NP	h	1.00	0.66	0.95	0.49	1.26	0.66	0.18	0.27	1.30
		$	2.15	2.15	2.15	2.15	4.29	2.15	2.15	2.15	4.29
10B	D2	h	**85.83**	**104.52**	**74.77**	**79.96**	**28.54**	**20.30**	6.00	5.22	0.86
		$	[9.89]	[12.08]	[8.62]	[9.20]	[3.33]	[2.42]	[0.81]	[0.69]	[0.12]
	D16	h	**29.27**	**40.31**	**66.97**	**53.08**	**28.54**	**20.30**	6.00	5.22	0.86
		$	27.60	37.72	61.64	49.68	26.68	19.32	6.44	5.52	0.92
	HB	h	**18.91**	**22.99**	**26.21**	**29.17**	**17.07**	**16.56**	5.05	5.22	0.77
		$	88.92	107.64	126.36	140.40	84.24	79.56	28.08	28.08	4.68
	NP	h	**9.99**	6.57	**9.46**	4.89	**12.57**	6.59	1.78	2.74	**12.97**
		$	21.45	15.02	21.45	10.72	27.89	15.02	4.29	6.44	27.89

Table 6.5 – Simulation cost using Microsoft Azure prices. We report estimated runtime in hours (**h**) and cost ($) for 1B and 10B clock cycles. Bold red hours exceed one workday (8 hours). We frame the lowest-priced configurations for easy identification. For long-running simulations, Manticore can finish 5/8 of the benchmarks in less than a workday, whereas other machines need 2–4 workdays.

Table 6.2 and then rounding to the next hour (Table 6.5). With longer runs, Manticore, in some cases, offers a lower cost than D16, despite its 2–18× higher base cost and unused resources.

Far more important, however, is the vast disparity in run duration. For the ten billion RTL cycle runs, Manticore finishes all of them in a long workday (13 hours). Multithreaded simulation requires up to two or more full days, while serial simulation can take most of a week. The productivity gain from several simulation runs per day dwarfs the minor cost savings (few dollars) from using a smaller machine: an engineer's time is much more valuable.

6.9 Global Stall

Manticore's performance is perfectly deterministic when all RTL state fits in on-chip SRAM. We now evaluate the impact of going off-chip and the cache's ability to mask its effect. Note that off-chip access occurs only if a benchmark contains a large RTL memory that does not fit in a single core's local scratchpad (32 KiB). RTL flip-flop state fits entirely within the distributed cores' register files and cannot cause off-chip accesses.

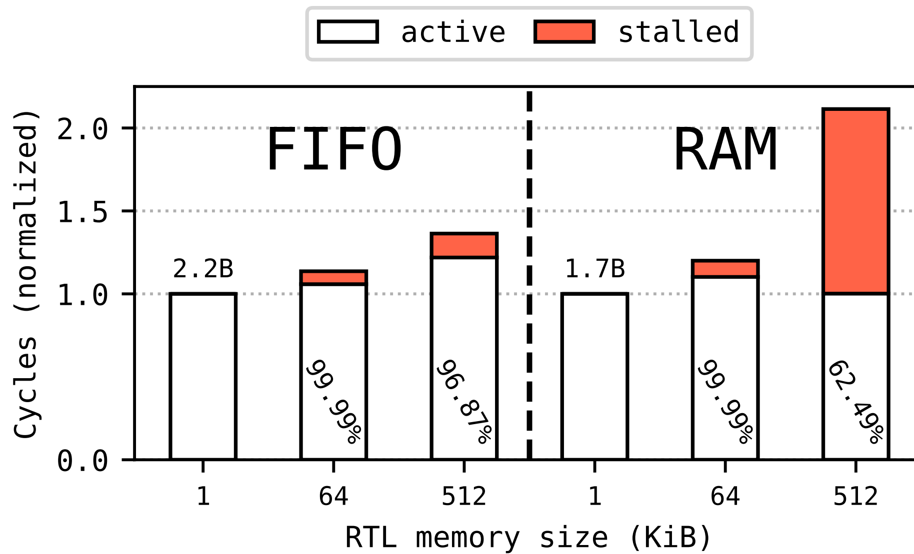


Figure 6.12 – Manticore's cache effectiveness. Number of machine cycles (lower is better) simulating a FIFO (left) and a RAM (right). White (orange) bars represent the number of cycles on which the compute clock is active (inactive). Numbers are normalized to the cycles needed to run the 1 KiB design. Cache hit rate is denoted inside each bar.

We use two RTL microbenchmarks running on a 1×1 Manticore grid at 500 MHz: (1) a FIFO, and (2) a RAM. The FIFO and RAM are sized at 1 KiB, 64 KiB, and 512 KiB. The FIFO reads/writes its memory sequentially, whereas the RAM accesses its memory with pseudo-random addresses (using a simple XOR-shift-128 generator). Each program runs for 16Mi RTL cycles and performs one load and one store operation per RTL cycle. We use hardware performance counters to log the total number of cycles, stalled cycles, cache hits, and cache misses. The 1 KiB configuration is a baseline for each microbenchmark since this memory fits in the scratchpad and incurs no global stalls. The 64 KiB represents a middle point where the state does not fit in the scratchpad but is entirely contained in the 128 KiB cache. Finally, the 512 KiB configuration corresponds to the scenario where the state is spread between the on-chip cache and off-chip DRAM.

Figure 6.12 reports the number of machine cycles needed to simulate the microbenchmarks, normalized to the number of cycles needed to simulate the 1 KiB designs. The white/orange region represents the number of active/stalled compute clock cycles. The numbers inside each bar denote the cache hit rate. Recall that a cache line is 256 bits wide and that global loads and stores move 16 bits of data. We measured DRAM latency to be ≈ 70 clock cycles (with a 475 MHz clock).

As expected, the 1 KiB designs are stall-free as the memories fit in a core's scratchpad and the cache is unused.

The 64 KiB designs experience an asymptotic cache hit rate of 1 as the RTL memory fits entirely within the cache. The FIFO populates the cache quickly, whereas the RAM needs more time since we use a random number generator to index it. As a result, the absolute number of cache misses

(and clock stalls) is higher for the RAM design and more FPGA cycles are needed to simulate 16Mi RTL cycles. Note that the orange bar on both 64 KiB designs look larger than 0.01% of the execution time. This is normal as each access to the cache in our prototype, even if the access hits, conservatively stalls the compute clock for *one* FPGA clock cycle (time to check if a hit or miss occurs). We could eliminate this conservative stall by sending out the cache address one cycle earlier in the pipeline, but decided to leave the core design unchanged to focus more on the compiler.

The 512 KiB FIFO has a high hit rate due to its spatial locality, so it incurs only a marginal number of stalls in aggregate. By contrast, large, randomly accessed RAMs run slower as the number of off-chip accesses increases.

6.10 Summary

We evaluated Manticore's performance, scalability, compile time, cost, and design decisions.

We compared Manticore's performance against Verilator, the fastest full-cycle RTL simulator, running on three high-end desktop and server x86 processors. Manticore is consistently faster than Verilator (serial and multithreaded) on all benchmarks that contain parallelism, despite using small slow cores that run at a fraction of the x86 machines' clock frequency.

While Verilator scales simulation to at most a few cores on our small benchmarks, Manticore continues to improve performance as the number of cores increases to 200–300. This performance scalability must be weighed against the single-core performance disparity between an x86 and Manticore's simple, low clock frequency cores.

Manticore's compile time is higher than Verilator's, but offers a faster time-to-result for hours long simulations. Its compile time can be improved through more extensive engineering.

We analyzed Manticore's cost in a cloud environment using the closest comparable FPGA and high-performance processors. While the hourly rate for renting an FPGA on the cloud is expensive, Manticore's speedup against high-performance x86 machines allows simulating a design in a fraction of the time. The end-to-end cost difference between Manticore and an x86 amounts to just a few dollars, an insignificant amount compared to the cost of an engineer who would otherwise need to wait a few more days for a simulation to run.

We separately evaluated each of Manticore's design decisions and their contribution to its performance. Manticore's BSP execution model requires duplicating instructions across cores to increase parallelism at the cost of more computation. The overhead of duplication increases with more cores, but is generally below 20–40% depending on the benchmark. Communication-aware partitioning is the most important step to achieve good parallel speedup on Manticore. While cores in most benchmarks exhibit relatively balanced workloads, some benchmarks do not and more work is needed to allow good parallel speedups for these on Manticore. Custom functions consistently improve performance, but do not specifically target a reduction in the critical path, and so are less useful than expected. Surprisingly, placement of code has little impact on end

performance as some cores are always compute-bound or port-limited at their NoC ingress port. A better NoC and a core with higher bandwidth at its ingress port are needed to better reap the benefits of placement algorithms. Manticore has abundant register capacity: its register files are highly underutilized and a smaller register file design would have been enough. This suggests that a re-design of the cores to increase core count would perhaps be beneficial to boost simulation capacity.

Finally, Manticore's current prototype allows at most 4096 machine cycles between synchronization points (the instruction memory size in each core). This clearly puts Manticore in the "small circuit" region of Figure 3.4, where performance scalability is *infeasible* on a general-purpose computer. If we are to improve simulation performance through parallelism, adding more cores to general-purpose processors will result in partitioned workloads that also fall in the "small circuit" region, so we need an architecture that can scale simulation when each processor has a small workload. Manticore's unconventional architecture avoids synchronization challenges and allows it to scale simulation over hundreds of cores irrespective of the workload size.

Part II

Low-level FPGA Manipulation Tools

7 A General Approach for Reverse-Engineering Xilinx Bitstream Formats

This chapter is about Bitfiltrator [56], a tool that allows modifying the bitstream of modern Xilinx FPGAs at the binary level. This work was originally conducted in the context of the Manticore project (see Part I). The goal was to program Manticore's instruction memories, data memories, register files, and custom function units at the bitstream-level so we could omit the bootloader from its design and simplify timing closure at high clock speeds. Unfortunately the FPGA primitives we used for the instruction and data memories (URAMs) cannot be programmed through the bitstream, and so we did not end up using Bitfiltrator to edit the bitstream of the final Manticore prototype.

We decided to spin off Bitfiltrator as educational material to teach others how to reverse-engineer parts of an FPGA's bitstream as this information was lacking in the literature. At the time when this project was conducted, Bitfiltrator was the only tool capable of editing the bitstream of large, multi-die Xilinx FPGAs, especially the specific device used in the Manticore project. Other tools have since added support for these devices [65], and so Bitfiltrator's differentiator remains its pedagogical contribution to the topic of bitstream manipulation.

7.1 Motivation

A bitstream is a binary file that contains the full or partial configuration of an FPGA. Xilinx's Vivado toolchain is currently the only way to generate a bitstream for the company's high-performance UltraScale and UltraScale+ devices. Unfortunately, bitstream generation is time-consuming as Vivado must load a placed and routed design checkpoint, then run design rule checks (DRC) to verify a design's legality before producing its bitstream. Designs that fail DRCs will not be generated to avoid damaging a device. Though Vivado allows selective disabling of DRCs before bitstream generation, the process can still take tens of minutes to load a design checkpoint for a large project before generating the bitstream. In short, bitstream generation is slow if Vivado is necessary. This raises the question of whether Vivado can be bypassed entirely in some circumstances?

Modifying an existing bitstream is one way to this end, and numerous projects have taken this path. Indeed, prior work has reverse-engineered parts of Xilinx's bitstream format to enable new

types of applications. For example, StateMover [5] demonstrated that a design running on an FPGA could be stopped, its bitstream read from the device, its user-visible state *extracted from the bitstream*, and this state then passed to a software simulator. This simulator continues the execution and exposes complete visibility into a design's state. After the software simulation, StateMover retrieves the design's state from the simulator and *embeds it back into the bitstream*. The FPGA is programmed with the new bitstream and continues executing the design at native speed. In addition, StateMover [5] manipulated a partial bitstream to disable LUTRAM masking during readback. Similarly, XBERT [46] employed knowledge of BRAM bits' locations in a bitstream to support APIs for zero-cost access to BRAMs on Xilinx FPGAs using partial reconfiguration. BitMan [75] reverse-engineered the bitstream format of parts of the interconnect and clocking network to re-reroute signal connections without using Vivado. Other work conducts fault injection experiments using FPGAs to study designs used in harsh environments [104]. Faults are emulated by directly modifying an FPGA's bitstream rather than re-generating it, to save time when studying numerous faults.

Applications clearly could benefit from the ability to modify an FPGA's bitstream directly, but how does one actually do this? While an FPGA's high-level configuration sequence may be documented [99], the binary format of the configuration data is not. It is unlikely a manufacturer will make this format public since a bitstream's binary format is tightly coupled to the proprietary physical layout of the underlying device [87]. Users who wish to manipulate a bitstream directly must then reverse-engineer its format.

The papers on prior *applications* of bitstream modification did not, however, explain *how* they reverse-engineered the bitstreams. This is understandable as space constraints limit a paper's ability to explain details. Some projects have made their source code available, which offers an executable description of how they modify a target bitstream. Unfortunately, this code contains device-specific constants of unclear origin. Using these codebases for other devices is further complicated by embedded assumptions that do not hold for other devices. To the best of our knowledge, BitMan [75] is the most advanced tool for bitstream manipulation as it can even convert FPGA bitstreams into a netlist [60]. However, BitMan is only distributed in binary form, and the publicly-available version does not support new, large multi-SLR devices[1]. In summary, much of the knowledge gained from past reverse-engineering projects is unavailable to learn from and apply to other devices.

Our work bridges this gap by focussing on the *methods* necessary to derive the fundamental parameters needed to implement a bitstream manipulation tool for Xilinx FPGAs. We explain why these parameters are needed, where they come from, and how to infer them. We also describe various pitfalls and erroneous device modeling assumptions to avoid. We implement an automated tool, Bitfiltrator, that uses a systematic algorithm to locate the precise position of a resource's initial configuration bits in a bitstream. We used Bitfiltrator on 40 UltraScale and UltraScale+ FPGAs to locate the configuration bits of initial LUT equations, LUTRAM and BRAM contents, and flip-flop (FF) values. Bitfiltrator does not require access to a physical FPGA to extract its device-specific parameters, and its source code is available [57]. Bitfiltrator is written

[1] Byteman [65] is the successor to BitMan and was released after Bitfiltrator. Byteman is open-source and supports more functionality than Bitfiltrator, but does not explain how the bitstream was reverse-engineered.

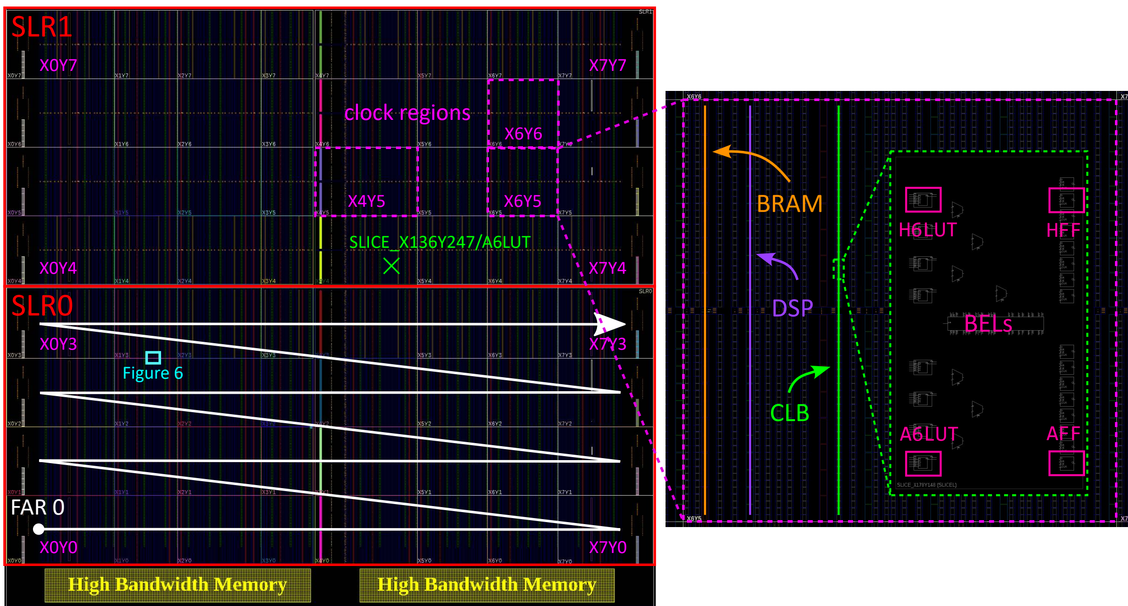


Figure 7.1 – Floorplan of an Alveo U50 datacenter FPGA. The left side of the figure shows the device's full floorplan with its two SLRs. We highlight three clock regions in magenta, and the LUT named SLICE_X136Y247/A6LUT in green. Frame addresses (see Section 7.3.1) start from 0 at the bottom-left clock region of each SLR and increase in the direction of the white arrow [75]. A zoomed-in image of the cyan region is shown in Figure 7.6 and will be explained in Pitfall 5. The right side of the figure zooms into clock region X6Y5. Clock regions have a columnar structure. We hightlight one BRAM column (orange), one DSP column (purple), and one CLB column (green). We further zoom into one CLB (called a SLICE) and show a subset of its BELs (LUTs and FFs).

entirely in Python and was developed and tested on x86-based processors. It can also run on ARM-based processors typically found in heterogeneous SoC-FPGA devices, and can therefore be used in autonomous or self-healing systems that rely on bitstream manipulation to correct errors.

The rest of this chapter is organized as follows: Section 7.2 presents background material on Xilinx FPGAs and their physical structure. Section 7.3 continues with a high-level description of a bitstream's organization. Section 7.4 describes how to locate a configuration frame in the bitstream using its addresss. Section 7.5 explains how to derive device-specific parameters needed to form a partial frame address for a major resource column in the FPGA. Section 7.6 shows how to (a) derive the architecture-specific parameters needed to expand the partial frame address into a complete one, and (b) map an individual resource in a major resource column to a specific bit in the target frame. Section 7.8 analyzes the derived parameters and draws insights into the physical implementation of Xilinx FPGAs. Section 7.7 experimentally verifies the derived parameters. Finally, Section 7.9 concludes.

7.2 Device Structure

Figure 7.1 shows an annotated floorplan of a Xilinx Alveo U50 datacenter FPGA, which we will refer to in the following sections. Xilinx FPGAs are hierarchical and consist of multiple super logic regions (SLR), each containing a grid of clock regions (e.g., X4Y5 in Figure 7.1). A clock region has a columnar structure where each column contains a homogeneous resource (CLBs, DSPs, BRAMs, etc.) and its associated clocking. Clock regions that differ in only their Y-offset (e.g., X6Y5 and X6Y6 in Figure 7.1) have the same width, but clock regions in a given row can have different widths (e.g., X4Y5 and X6Y5 in Figure 7.1). All clock regions have the same height. The height of a clock region in UltraScale and UltraScale+ devices is 60 CLBs, 24 DSPs, and 24 18Kb BRAMs [101].

Vivado identifies cells in a design by a basic element (BEL) name. BELs are named using a *resource-specific* prefix and XY coordinate system (e.g., `SLICE_X136Y247/A6LUT` in Figure 7.1). BELs have *properties* that Vivado uses to set specific configuration bits when generating a bitstream. For example, LUTs have a 64-bit `INIT` property that configures their equation.

Our goal is to map the `INIT`-related properties of a specific BEL (LUT, LUTRAM, FF, or BRAM) to bit offsets within the bitstream. Pitfall 1 explains a naive, unstructured method to approach this task and its consequences. It then outlines a principled, structured way of obtaining the bit offsets of interest, which the rest of the chapter explains in detail.

Pitfall 1: An unstructured reverse-engineering approach is impractical

The apparently most straightforward way to reverse-engineer the format of BELs in the bitstream is to use binary analysis: Compare the bitstream obtained from an empty design checkpoint against that obtained from a minimally-edited one (e.g., by *individually* setting a BEL's properties using Vivado Tcl commands). One might expect that the differing bits are related to the resource of interest.

However, this naive approach has two significant shortcomings. First, toggling even a *single* bit in a BEL property may cause multiple changes to the bitstream. These changes are often at distant—seemingly unrelated—locations, which makes it difficult to identify the bits that are actually influenced by the BEL property being studied. Second, bitstream generation is controlled by multiple configuration options. These options directly influence the offsets at which bits are located in the bitstream. Any *absolute* bit offsets found in a given bitstream are therefore unusable for bitstreams generated with different configuration options (e.g., partial bitstreams).

A better way to tackle the reverse-engineering task is to employ a structured approach that relies on understanding how the bitstream configures the FPGA. Two key questions need to be answered: First, what is the smallest granularity at which configuration is performed? Second, how are BELs related to this granularity? Answering these two questions allows us to locate an atomic unit of configuration, and then study its contents

to extract *relative* bit offsets to device-specific features. This allows transfering common bit offsets learnt from one device to another.

7.3 Bitstream Structure

Xilinx bitstreams are composed of two parts: a text header and a binary section. The header is a key-value store of metadata about the bitstream [69], and the binary section contains the bitstream data. The binary section is parsed by first locating a unique byte pattern called the SYNC_WORD [99]. Data after the SYNC_WORD consists of a sequence of packets that write to the internal registers of a configuration processor. Packets are composed of operations until a write to the CMD register occurs with a payload of DESYNC [99]. The process repeats until all packets are consumed.

7.3.1 Single-SLR Configuration

The FPGA's configuration processor has 32 registers, but the configuration-related packets in the bitstream write to only two relevant registers: FAR, and FDRI.

FAR register

The smallest configurable unit in Xilinx FPGAs is called a *frame*. Every frame has a unique SLR-local address composed of a 4-tuple: a block type[2] (CLB_IO_CLK or BRAM_CONTENT), a major row, a major column, and a minor column (see Figure 7.2). In the bitstream, the Frame Address Register (FAR) stores the current frame's address. Frames with a block type of BRAM_CONTENT contain the initial contents of BRAMs in the device. All other device resources (CLBs, DSPs, IOs, interconnect, etc.) are configured by frames with a block type of CLB_IO_CLK. All frames in an architecture have the same size and span one element (CLB, BRAM, DSP, etc.) horizontally with a height of one clock region [103].

FDRI register

A frame at address FAR is configured by writing a frame-sized payload to the Frame Data Input Register (FDRI). Multiple consecutive frames can be configured with a single, large write to FDRI. The configuration processor *auto-increments* the contents of FAR at every frame-sized interval. Frame addresses in every SLR start from 0 and increase following the order defined by the white arrow in Figure 7.1: minor column, major column, major row, and block type [75].

A typical full bitstream contains a *single* write to FAR followed by a device-sized write to FDRI with all configuration frames. In contrast, a typical partial bitstream contains *multiple* small, scattered writes to these registers and covers a subset of the device's frames.

[2] The block type field is 3 bits wide, but only two values are legal: (1) 0b000 for CLB_IO_CLK, and (2) 0b001 for BRAM_CONTENT.

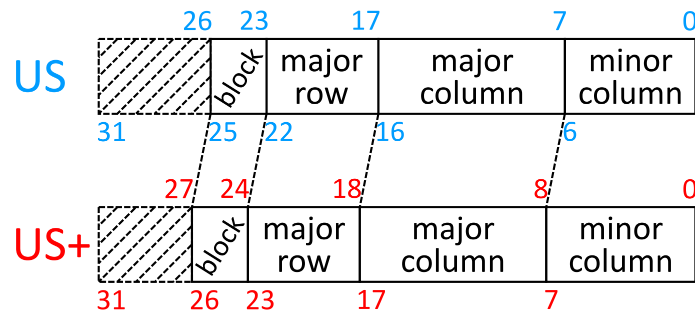


Figure 7.2 – The frame address format in UltraScale and UltraScale+ devices. The address is composed of a 4-tuple: block type, major row, major column, and minor column [99]. The block type is either CLB_IO_CLK or BRAM_CONTENT. The number above/below each field marks its start/end offset in the 32-bit FAR word. UltraScale+ devices have an expanded minor column field compared to UltraScale devices.

7.3.2 Multi-SLR Configuration

Large FPGAs, like the U50 shown in Figure 7.1, are not monolithic chips. Such devices are instead composed of multiple SLRs. Each SLR is essentially a small, independent FPGA inside a larger device. As a result, each SLR is configured separately by the bitstream. The configuration processor's IDCODE register is used for this task.

IDCODE register

Writes to the IDCODE register identify the target device to be configured. Writes to the FDRI register must be preceded by a write to IDCODE. The bitstream of a small, single-SLR device contains one write to the IDCODE register, whereas the bitstream of a large multi-SLR device writes a value to IDCODE to configure a specific SLR.

The architecture configuration manual [99] lists the idcodes for a subset of the UltraScale and UltraScale+ FPGAs. However, only *one* idcode is provided per FPGA, i.e., the idcodes of individual SLRs are not reported. The SLR idcodes therefore need to be extracted from a bitstream.

Listing 7.1 details the high-level configuration-related packets in the full bitstream of an Alveo U200 datacenter accelerator card. The U200 is a large device with three identically-sized SLRs configured independently by changing IDCODE. Pitfall 2 continues with a description of how SLRs are ordered in the bitstream.

Pitfall 2: SLR ordering in the bitstream

The order of writes to the IDCODE register do *not* match the bottom-to-top order of SLRs (i.e., SLR0, SLR1, SLR2) shown in the Vivado GUI! The U200 above, for example, configures SLR1 before it configures SLR0 and SLR2.

We use an SLR object's CONFIG_ORDER_INDEX property (not to be confused with SLR_INDEX) from Vivado to recover the order in which SLRs must be configured in a multi-SLR device.

```
# SLR 1
PKT_TYPE = TYPE1, OP = WR, REG = IDCODE, WORD_COUNT =       1), PAYLOAD = { 0x04b37093 }
PKT_TYPE = TYPE1, OP = WR, REG = FAR   , WORD_COUNT =       1), PAYLOAD = { 0x00000000 }
PKT_TYPE = TYPE1, OP = WR, REG = CMD   , WORD_COUNT =       1), PAYLOAD = { WCFG }
PKT_TYPE = TYPE1, OP = WR, REG = FDRI  , WORD_COUNT =       0), PAYLOAD = {  }
PKT_TYPE = TYPE2, OP = WR, REG = FDRI  , WORD_COUNT = 6679260), PAYLOAD = { ... }

# SLR 0
PKT_TYPE = TYPE1, OP = WR, REG = IDCODE, WORD_COUNT =       1), PAYLOAD = { 0x04b22093 }
PKT_TYPE = TYPE1, OP = WR, REG = FAR   , WORD_COUNT =       1), PAYLOAD = { 0x00000000 }
PKT_TYPE = TYPE1, OP = WR, REG = CMD   , WORD_COUNT =       1), PAYLOAD = { WCFG }
PKT_TYPE = TYPE1, OP = WR, REG = FDRI  , WORD_COUNT =       0), PAYLOAD = {  }
PKT_TYPE = TYPE2, OP = WR, REG = FDRI  , WORD_COUNT = 6679260), PAYLOAD = { ... }

# SLR 2
PKT_TYPE = TYPE1, OP = WR, REG = IDCODE, WORD_COUNT =       1), PAYLOAD = { 0x04b24093 }
PKT_TYPE = TYPE1, OP = WR, REG = FAR   , WORD_COUNT =       1), PAYLOAD = { 0x00000000 }
PKT_TYPE = TYPE1, OP = WR, REG = CMD   , WORD_COUNT =       1), PAYLOAD = { WCFG }
PKT_TYPE = TYPE1, OP = WR, REG = FDRI  , WORD_COUNT =       0), PAYLOAD = {  }
PKT_TYPE = TYPE2, OP = WR, REG = FDRI  , WORD_COUNT = 6679260), PAYLOAD = { ... }
```

Listing 7.1 – Alveo U200 bitstream dump. We omit non-configuration-related packets for brevity. The U200 is a large FPGA with three SLRs. Note that the SLRs are not configured in order.

Matching the sequence of idcodes in a complete bitstream against the SLR configuration indices associates the SLRs with their idcodes.

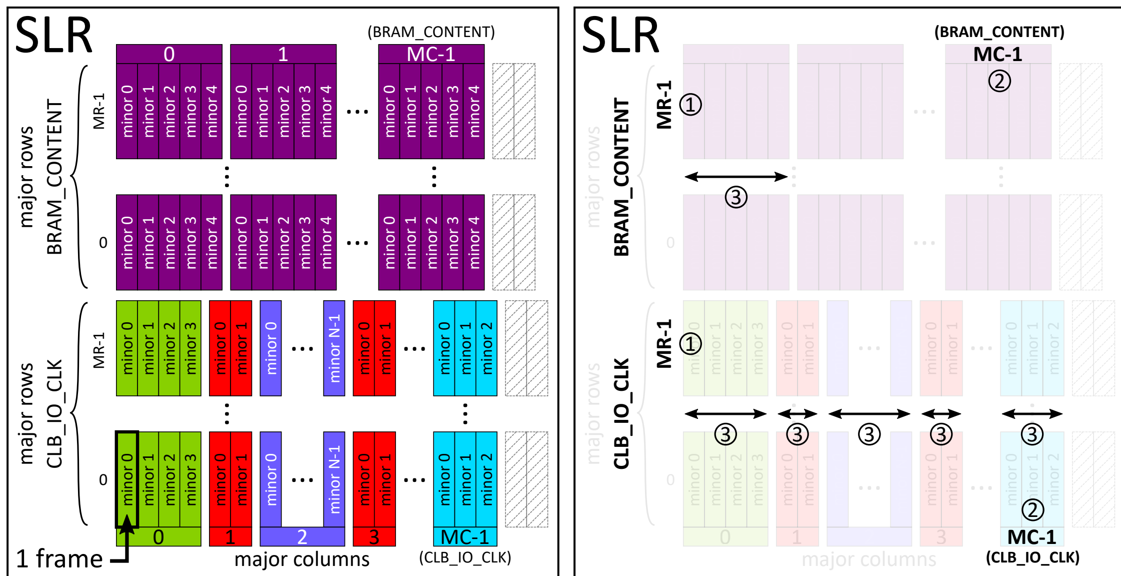
7.4 Locating a Frame by Address

Frames are identified by SLR-local addresses (see Section 7.3) and we know how to locate the configuration frames of a given SLR in the bitstream (see Pitfall 2). What remains is to locate a frame, using its address, among the configuration frames of a given SLR.

7.4.1 Frame Addressing Scheme

Frames in a full bitstream are written as a single SLR-sized array to the FDRI register with a base frame address of 0. In contrast, frames in a partial bitstream are written as multiple smaller arrays to the FDRI register with different base frame addresses. In both cases, similar to the FPGA's configuration processor, a frame can be associated with its address by keeping track of the base frame address seen when parsing a write to FAR in the bitstream, and incrementing this base address at frame-sized intervals until the array is consumed.

Since a frame address consists of multiple fields (see Figure 7.2), we need to know the boundaries of each field to increment the address correctly. We first describe some constraints that allow us to map the various fields in the frame address to physical characteristics of the target FPGA. This will then allow us to infer the fields' boundaries. The constraints are the following:



(a) Frame address field boundaries. Major row indices $[0, \ldots, MR)$ are shown on the left, major column indices $[0, \ldots, MC)$ indices are shown at the top and bottom, and minor column indices are shown in each rectangle.

(b) The device-specific bitstream parameters needed to locate a frame by its address: (1) the number of major rows, (2) the number of major columns per row, and (3) the number of minors per major column.

Figure 7.3 – The SLR frame indexing scheme. Every color represents a homogeneous resource such as a CLB column, DSP column, etc. Frames of each block type (CLB_IO_CLK or BRAM_CONTENT) with the same major column index have the same number of minor columns, otherwise the column would not contain a homogeneous resource. The number of minors in each major column are for illustration purposes only and do not reflect the quantity in a device. The two empty frames at the end of each major row are discussed in Pitfall 4.

- Every frame spans one clock region vertically [103], so the major row field in the frame address must correspond to the Y-offset of the clock region in which the BEL is located. The Y-offset is *relative* to the lowest row of the SLR as frame addresses start from 0 in each SLR.
- The frame address structure reveals that every major column contains multiple minor columns. Any two major columns that contain the same resource (CLBs, DSPs, BRAMs, etc.) must have the same number of minor columns; otherwise, the resources would not be homogeneous. The major column field in the frame address therefore must identify a resource column.
- The block type field is located in the upper bits of the frame address and its legal values are `0b000` for CLB_IO_CLK and `0b001` for BRAM_CONTENT. As a result, though BRAM columns in a device may be physically adjacent to those of CLBs, DSPs, etc., the frame address structure shows that their contents are stored after CLB_IO_CLK frames.
- The number of major columns with block type BRAM_CONTENT must match the number of BRAM columns in the device. This number is necessarily smaller than the number of major columns with block type CLB_IO_CLK as there are significantly fewer BRAM columns in a device than the total resource columns.

Given these properties, the left side of Figure 7.3 illustrates how frames of an SLR are ordered in the bitstream. The right side of Figure 7.3 highlights the device-specific parameters needed to locate a configuration frame from its address: (1) the number of major rows, (2) the number of major columns per row, and (3) the number of minors per major column. These three values must be derived for both the CLB_IO_CLK and BRAM_CONTENT block types. We tackle these points next.

7.4.2 Enumerating Frame Addresses

Enumerating all valid frame addresses in a device is the simplest way of inferring the device-specific parameters introduced in Section 7.4.1. All valid frame addresses in a device exist only in a full bitstream. Therefore, our discussion below assumes a full bitstream, not a partial one, is being analyzed.

In general, the bitstream does not contain the valid addresses as the operations write the FAR register only once with a base frame address of 0 and then perform a single, large write to the FDRI register. The configuration processor auto-increments the frame address at frame boundaries without subsequent inputs.

Devices, fortunately, often have additional features that can help with reverse engineering. In particular, Vivado is capable of generating bitstreams with per-frame CRCs [102]. Such bitstreams have two desirable properties: First, each frame is loaded individually. Second, every write to FDRI is followed by a write to the CRC register. The address of the next frame is then *explicitly* written to the FAR register. We can therefore obtain all valid frame addresses for each SLR with the following procedure:

1. Generate a bitstream with per-frame CRCs for a target FPGA.
2. Parse the bitstream's binary section and extract packets that write to the FAR register.
3. Filter out frame addresses with a reserved block type[3].

With this list of valid frame addresses, we sort the addresses in increasing order and identify the points at which each field (see Figure 7.2) in the frame address wraps around. This provides a basis for computing the number of major rows, major columns per row, and minor columns per major column for both CLB_IO_CLK and BRAM_CONTENT block types. We can now locate a frame in the bitstream by its address, as we can correctly increment the frame address at every frame-sized write to the FDRI register in the bitstream.

Enumerating all valid frame addresses uncovers two further bitstream navigation pitfalls, which we discuss in Pitfall 3 and Pitfall 4.

[3]This step is needed as, once all frame configurations are written to FDRI, the bitstream launches the FPGA startup sequence by writing to the CMD register. This write to CMD is followed by a write to FAR with a reserved block type of 0b111. This write to FAR is irrelevant to the configuration of the FPGA as it comes after all writes to FDRI, hence we filter it out.

Pitfall 3: Hidden major rows in devices

The number of clock regions visible vertically in the Vivado GUI is *not* always equal to the number of major rows in the device! For example, the smallest member of the Kintex UltraScale family (xcku025) has three rows of clock regions, but extracting its frame addresses using the method in Section 7.4.2 reveals it is, in fact, composed of five rows.

The next larger member of the Kintex UltraScale family (xcku035) also has five rows of clock regions, and the frame addresses of both devices are identical (at least at the binary level). It is likely that the smaller device is simply a restricted version of the larger one. This observation may hold for other devices.

Pitfall 4: Empty frames at the end of a major row

In addition to bitstreams with per-frame CRCs, Vivado also supports generating a *debugging bitstream*. A debugging bitstream also loads each frame individually, and every write to the FDRI register is followed by a write of the current frame address to the LOUT register. Writing to the LOUT register drives data to the DOUT pin of the FPGA in a serial daisy-chain configuration and is useful for debugging how far the device has advanced into its configuration.

Interestingly, the number of frames in a debugging bitstream is less than the number of frames in a standard bitstream. Creating a design where a majority of BELs on an FPGA is populated, emitting both a full and debugging bitstream from the same design checkpoint, and comparing the frames reveals that there are always two empty frames at the end of a major row in a full bitstream.

One therefore must take these two empty frames into account when incrementing frame addresses at row boundaries. We believe the frame addresses emitted by the debugging bitstream are the “real” ones used in a design and the two empty frames at the end of each major row are likely an implementation detail.

Note that debugging bitstreams exist only for standard FPGAs [102], not SoC-FPGA devices. Nevertheless, we experimentally confirmed that the latter also exhibit these two empty frames at the end of each major row.

7.5 Extracting Device Parameters

We can locate any frame in a bitstream with a valid frame address. We now discuss our systematic approach to map a BEL to an SLR idcode and a *partial* frame address using device-specific parameters. Section 7.6 further develops this approach and will explain how to obtain and use architecture-specific parameters to get the *full* frame address and frame offset for each bit in a BEL’s INIT property.

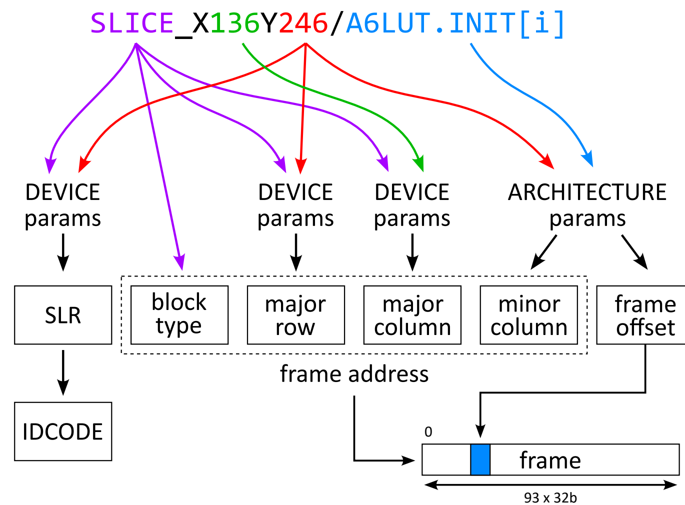


Figure 7.4 – The process for mapping a bit in a LUT's INIT property to a bit in the bitstream. We use the frame size of UltraScale+ devices.

Our method is hierarchical and is outlined in Figure 7.4. We first explain a shortcut for obtaining the major column numbers of CLBs and BRAMs. We then explain a general approch for mapping *any* resource to a major column number. Finally, we detail how to compute the SLR idcode, block type and major row of a BEL to form the partial frame address.

It is essential to disable bitstream encryption and compression in Vivado [102] to permit binary analysis before undertaking any of the following steps.

7.5.1 Mapping Resource Columns to Major Column numbers

Shortcut (for CLB and BRAM columns)

Xilinx devices support *readback capture* [95], a mechanism to extract a bitstream from a programmed FPGA and identify the *user-state* bits of the underlying design. A user-state bit is a user-configured memory (flip-flop, LUTRAM, BRAM, or BRAM output register). Note that standard LUTs are not user-state bits as the user cannot change their value at *runtime*. As the bitstream format is undocumented, Vivado generates a *logic location file* alongside the bitstream, which provides the bit position of all user-state bits in the design. Figure 7.5 shows a short excerpt from a logic location file. In addition to the bit positions, the file conveniently contains the frame address of instantiated state bits. We then extract the major column numbers from the frame addresses reported in the file.

We therefore obtain the major column numbers of CLBs and BRAMs with the following procedure:

1. Iterate over *all* clock regions (see Pitfall 5);
2. Iterate over all resource columns of the clock region;
3. Place one instance of a resource in every column that has the resource of interest;
4. Generate a bitstream and its accompanying logic location file;

Figure 7.5 – Logic location file excerpt. The design is that of a small UltraScale+ FPGA (xazu1eg). The logic location file details two flip-flop bits, two LUTRAM bits, and two BRAM data bits. The details reported for each entry include: (1) the target SLR, (2) the frame address, and (3) the frame offset (see Section 7.6). The major column of a resource can be extracted from its frame address.

5. Parse the logic location file and extract major column numbers from the reported frame addresses.

Extracting CLB major column numbers is done with a design that contains a flip-flop in every CLB column as LUTs, LUTRAMs, and flip-flops are all contained in CLBs and therefore share the same major column [103]. Similarly, BRAM major column numbers can be determined by instantiating any block-based memory implemented using BRAMs (FIFO, etc.) in every BRAM column.

Pitfall 5: Virtual device-high CLB columns are not all fully populated

Single-SLR devices have homogeneous resource columns in each major row. Determining the resource columns in *one* row is therefore sufficient to know the resource columns in *all* other rows. However, this property does not hold for multi-SLR devices.

In multi-SLR devices, major rows at the boundary of two SLRs contain special LAGUNA tiles that enable signals to cross between SLRs. The LAGUNA tiles take up space previously occupied by a virtual device-high CLB column (see Figure 7.6). In other words, SLR-boundary rows contain fewer CLB resources than SLR-internal rows.

Care must be taken when looking up a CLB's major column number in SLR-boundary rows to avoid erroneous frame indexing. The method described in Section 7.5.1 yields the exact major column number for all CLBs in a device, regardless of the clock region row in which they are located, as we place a resource by iterating over the resource columns of *all* clock regions.

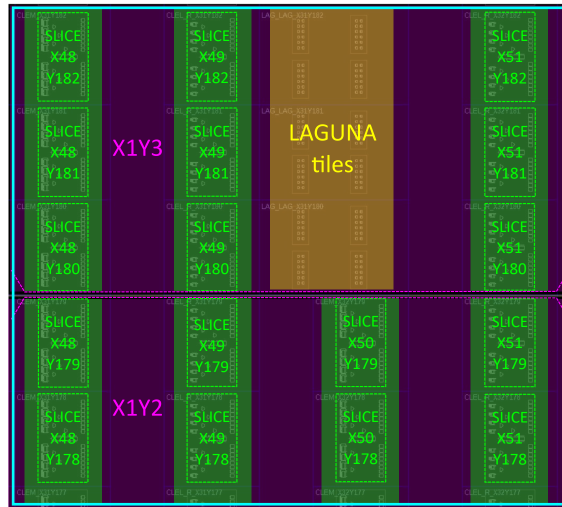


Figure 7.6 – Zoomed-in region in cyan from Figure 7.1. We see a small section of the last major row in SLR0 before the SLR0–SLR1 boundary. The LAGUNA tiles in SLR-boundary rows enable inter-SLR connections and take up space previously occupied by CLB column `SLICE_X50Y*` in SLR-internal rows. As a result, the CLB range `SLICE_X50Y180`–`SLICE_X50Y239` does *not* exist in the device and major row 3 in SLR0 has physically fewer CLB columns than the other major rows.

General approach (illustrated with DSP columns)

One way to obtain the major column number of any resource is to use binary analysis: By comparing the bitstream of a carefully-crafted design against an empty[4] one, we can identify frames that differ and extract the major column numbers from their addresses.

It is important to create a minimal design to reduce the use of auxiliary structures in the FPGA that will influence the generated bitstream. We use a similar design process as in Section 7.5.1 to achieve this: (1) Iterate over all clock regions, (2) place *one* instance of a resource in every column that has the resource of interest, and (3) generate a bitstream. Comparing the generated bitstream against the an empty one will reveal multiple differing frames. In theory, the major columns of the differing frames are those containing the resource of interest. In practice, some auxiliary FPGA structures are inevitably used, so we must filter out the frames they influence in the bitstream to avoid inferring major column numbers from frame address that do not contain the target resource. This is detailed in Pitfall 6.

Pitfall 6: Filtering out CLB and interconnect noise when isolating a resource

Crafting a minimal design to reduce differences when comparing bitstreams is challenging as a design must satisfy multiple constraints for a bitstream to be generated. One such constraint is that *BEL input pins must be driven* (BEL output pins can be left unconnected though). In practice, this means using constants (usually 0) to drive the BEL's input pins.

[4] Note that Vivado will not generate a bitstream for an entirely empty design. A near-empty proxy is a single register with its inputs connected to 0.

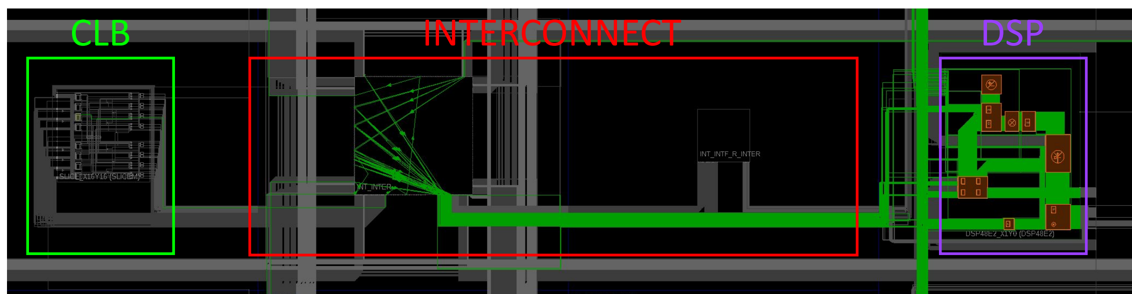


Figure 7.7 – CLB and interconnect noise in a bitstream. The design contains an isolated DSP whose inputs are tied to 0 and whose outputs are unconnected. A CLB is used to generate the constant and the interconnect is used to route it to the DSP's input pins.

However, arbitrary resource columns may not be able to generate these constant inputs. In such cases, the constants are driven by CLBs and are *routed* to the target BEL. Figure 7.7 illustrates this case with an isolated DSP whose inputs are tied to 0 and whose outputs are unconnected.

Since the FPGA's CLBs and interconnect are used, their configuration affects the bitstream. Naively comparing the bitstream crafted in Section 7.5.1 against an empty bitstream would reveal that frames differ in major columns of multiple resources, not just in the major columns that contain DSP resources. We must therefore filter out unwanted columns.

Filtering out CLB columns is simple as we derived the device-specific major column numbers of CLBs following the procedure described in Section 7.5.1. We can therefore remove any differing frame whose major column number matches a CLB column.

We use an assumption to filter out interconnect columns: An FPGA's highly-flexible interconnect columns require significantly more frames to configure than a DSP. We know the number of frames needed to configure different major columns since we derived the number of minor columns they each contain (see Section 7.4.2). We can therefore easily differentiate interconnect columns from DSP ones. For example, in UltraScale+ devices, we see that differing frames are in major columns that contain either 8 or 76 minors. We infer that DSP major columns are those with 8 minors.

7.5.2 Putting it Together: Forming a Partial Frame Address

We now illustrate how to construct the idcode and partial frame address for a specific BEL. We use the 6-LUT from Figure 7.1 (SLICE_X136Y247/A6LUT) as an example.

We start by determining the idcode whose frames we must search. We know the SLRs in the U50 are 4 clock regions high (see Section 7.4.2) and that the height of each clock region is 60 CLBs (see Section 7.2). We therefore infer that SLICE_X136Y247 is located in SLR1 (i.e., $\lfloor 247/(4 \times 60) \rfloor = 1$). We now know its idcode (see Pitfall 2).

Next, we identify the block type of the CLB. We know CLBs are not initial BRAM contents, so the block type must be `CLB_IO_CLK`.

Then we compute the major row of the CLB. We again use our knowledge of a clock region's height to infer that the CLB is located in *absolute* row major 4 (i.e., $\lfloor 247/60 \rfloor = 4$). This corresponds to major row 0 when indexing is relative to SLR1.

Finally, we obtain the major column. We know the major column indices for a given resource in every row (see Section 7.5.1). Looking up the 137$^{\text{th}}$ entry of the CLB major column indices in major row 0 yields the major column number of the CLB, which is major column 262 in the U50.

7.6 Extracting Architecture Parameters

The work to this point derives the device-specific parameters needed to isolate a BEL to a subset of the configuration frames. This subset is formed from three of the four elements in a frame address: (1) block type, (2) major row, and (3) major column.

The remaining parameter is the minor column and frame offset of each bit in a BEL's `INIT` property within a major resource column. The minor column and frame offset are *architectural* parameters since the structure of a specific resource column is identical among all devices that share a common architecture.

7.6.1 BRAM Format

We start by selecting a BRAM column and create a design that populates *all* its BELs. A clock region is 24 18Kb BRAMs high, which amounts to a total of 422 368 bits. Given the long time to generate a bitstream, no fuzzing-based approach can practically enumerate every bit's minor and frame offset in any reasonable amount of time.

Fortunately, BRAM contents are user-state bits, so we again use the logic location file shortcut presented in Section 7.5.1 here to make Vivado do the work and describe the frame addresses and offsets of all 422 368 bits in one step. Parsing the logic location file reveals the minor columns and frame offsets.

7.6.2 CLB Format

Resource columns are formed of *tiles*. CLBs are found in four types of tiles[5]: `CLEL_L`, `CLEL_R`, `CLEM`, `CLEM_R`. In theory, the BELs in each type of tile could have a different format in the bitstream, so the experiments below need to be run for all four types of tile columns[6]. Like BRAMs, the experimental setup here consists of an entire column of a single tile type populated with LUTRAMs (if applicable for the subset of CLBs containing LUTRAMs), LUTs, and flip-flops.

[5] Their names differ slightly between UltraScale and UltraScale+ architectures, but they serve the purpose.

[6] All tiles that contain BRAMs are of the same type; hence we need not try multiple tile types to extract a BRAM's format in Section 7.6.1.

Flip-flops and LUTRAMs

Discovering the minor columns and frame offsets of flip-flops and LUTRAMs is simple as the tools consider these BELs to be user-state bits. We again use the logic location file to determine their format in a column quickly.

Standard LUTs

Determining the minor columns and frame offsets of standard LUTs is more complicated as these BELs are not user-state bits, so no logic location file is available. We want to discover the format of the 64-bit LUT equation for each LUT in a column. There are 480 6-LUTs in a column, which amounts to a total of 30 720 configuration bits. Though an order of magnitude smaller than the number of BRAM bits in a column, this number is large enough to make a column-high fuzzing-based approach impractical, so another approach is needed.

We use the assumption that hardware is highly regular to reduce the search space: It is reasonable to expect all tiles of the same type to have the same format in the bitstream, with different offsets. We, therefore, only need to fuzz an individual tile in the CLB column. Each tile contains eight 6-LUTs and thus $8 \times 64 = 512$ configuration bits, a tractable number to fuzz. Fuzzing requires generating 512 bitstreams that try all *one-hot* `INIT` configuration bits in a tile. Comparing the fuzzed bitstream against a baseline will reveal the minor column and frame offset of the tile's configuration bits.

Fuzzing the `INIT` property of a CLB tile requires creating a baseline bitstream against which we can compare fuzzed variants. It is essential to create the design checkpoint post-place and route to ensure all future modifications of a LUT's `INIT` property do not modify routing and create additional noise when comparing the fuzzed bitstream against the baseline. The baseline bitstream is created as follows:

1. Select a CLB with the target tile type of interest;
2. Populate all 8 LUTs in the CLB (set their `INIT` property to 0, mark them as `DONT_TOUCH`, set their inputs to 0, and disconnect their outputs);
3. Place and route the design;
4. Generate a baseline design checkpoint and bitstream.

The fuzzed bitstreams are then obtained by loading the baseline design checkpoint and setting a *single* bit to 1 in the 8 LUT equations by manipulating the LUT's `INIT` property in Vivado. Comparing these bitstreams against the baseline will yield the minor columns and frame offsets of all 512 LUT configuration bits in a tile. Translating the minor column and frame offsets of the LUTs from the fuzzed tile to the other tiles in the column requires only locating an anchor bit in each tile. A simple anchor bit to use is the same flip-flop (e.g., `AFF`) in every tile as we already determined all their minor columns and frame offsets (see Section 7.6.2).

INIT [i]	SLR	Block Type	Major Row	Major Column	Minor Column	Frame Offset
0	SLR1	CLB_IO_CLK	0	262	11	351
1	SLR1	CLB_IO_CLK	0	262	10	351
2	SLR1	CLB_IO_CLK	0	262	9	351
3	SLR1	CLB_IO_CLK	0	262	8	351
4	SLR1	CLB_IO_CLK	0	262	11	350
5	SLR1	CLB_IO_CLK	0	262	10	350
6	SLR1	CLB_IO_CLK	0	262	9	350
7	SLR1	CLB_IO_CLK	0	262	8	350
...	...	...	...	...	...	...
60	SLR1	CLB_IO_CLK	0	262	11	336
61	SLR1	CLB_IO_CLK	0	262	10	336
62	SLR1	CLB_IO_CLK	0	262	9	336
63	SLR1	CLB_IO_CLK	0	262	8	336

Table 7.1 – Frame addresses and offsets of SLICE_X136Y247/A6LUT on a U50 FPGA. The minor column and frame offset of the configuration bits follow well-defined patterns.

7.6.3 Putting it Together: Forming a Full Frame Address

Having discovered the minor and frame offset of each bit in the INIT property of LUTs, LUTRAMs, BRAMs, and flip-flops, we are now able to form a complete frame address for any such BEL and locate it in the bitstream. We continue the 6-LUT example we partially formed in Section 7.5.2 and seek to form a frame address for bit 0 in its LUT equation (i.e., SLICE_X136Y247/A6LUT.INIT[0]).

We already know this LUT has block type CLB_IO_CLK, and is in SLR1's major row 0 and major column 262. What remains is to determine the minor column and frame offset. The procedure detailed in Section 7.6.2 reveals that SLICE_X<>Y247/A6LUT.INIT[0][7] is in minor column 11, at frame offset 351.

Table 7.1 extends our systematic approach to subsequent bits of the BEL's INIT property. Well-defined patterns can be seen for the minor column and frame offset of the configuration bits. One can then construct formulas to directly index a bit in the bitstream given its name. The same technique can be used for other resources of interest (BRAMs, LUTRAMs, flip-flops, etc.).

Knowledge of the SLR, full frame address, and frame offset for a given bit allows a bitstream manipulation tool to read or write it.

[7]The CLB's X-coordinate does not influence the minor column and frame offset, so we omit it here.

Architecture	Vivado Name	Count
UltraScale	Kintex UltraScale	2 / 12
	Virtex UltraScale	0 / 7
UltraScale+	Kintex UltraScale+	6 / 10
	Virtex UltraScale+	8 / 31
	Zynq UltraScale+	24 / 38
	Zynq UltraScale+ RFSOC	0 / 16

Table 7.2 – Summary of devices on which Bitfiltrator was tested. Devices are enumerated in Vivado and categorized by their architecture and family name. Bitfiltrator was only tested on devices for which bitstreams can be generated using the free Vivado "WebPack" license. No Virtex UltraScale or Zynq UltraScale+ RFSOC devices are available in the free version of Vivado.

7.7 Evaluation

We implement the techniques described in Section 7.4, Section 7.5, and Section 7.6 in Python (for bitstream parsing and analysis) and Tcl (for replicating and configuring BELs in Vivado). Bitfiltrator only takes as input a target FPGA part number. No XDC constraint file is needed as none of the experimental setups use I/O pins. All experiments are conducted on a machine with an Intel Xeon E5-2680 v3 processor running Vivado 2022.1.

We use Vivado to enumerate *all* UltraScale/UltraScale+ part numbers and apply our automated flow to devices which do not require a full Vivado implementation license (devices included with the free "WebPack" license). Table 7.2 categorizes the devices by their architecture name[8]. Note that the Virtex UltraScale+ devices are all members of the Alveo datacenter-grade device family.

The device-specific parameters extracted are: (1) the number of visible and "hidden" major rows (see Pitfall 3), (2) the number of major columns in every row, (3) the number of minor columns in every major column, and (4) the major columns of CLBs, BRAMs, and DSPs. We use the smallest member of each device family as a proxy to extract the architecture-specific parameters (minor columns and frame offsets) of LUT, LUTRAM, BRAM, and flip-flop `INIT` properties to reduce bitstream sizes while fuzzing. Device-specific parameters are extracted in parallel using 12 cores. Similarly, the `INIT` property of the 8 LUTs in a CLB tile are fuzzed in parallel with 12 cores, then translated to all other LUTs in a CLB column. The total end-to-end runtime for extracting the device- and architecture-specific parameters for all 40 devices is approximatively 3 hours.

We compared the discovered architectural parameters against those from project U-Ray [4], a project that reverse-engineers the bitstream format of UltraScale and UltraScale+ FPGAs to support the development of open-source FPGA toolchains. Project U-Ray provides databases that contain the minor column and frame offset of the configuration bits of *tiles* in UltraScale+

[8] The architecture names are obtained from Vivado, not marketing materials, as devices' categorization differs between both sources. The following Tcl command is used to extract the full architecture name:
`get_property ARCHITECTURE_FULL_NAME ${part}`.

FPGAs. Our minor columns and frame offsets for LUT equations, LUTRAM and BRAM contents, and register values match those from U-Ray.

However, project U-Ray only provides a per-tile database of minors and frame offsets, and does not offer translated versions of these numbers for the other tiles of a resource column. Additionally, the databases only cover UltraScale+ devices, not UltraScale ones.

We, therefore, conduct an additional experiment to validate all of our device-specific and architectural parameters. We first populate *all* LUT, register, and BRAM BELs in a device. Then, we use a Tcl script to set the INIT property[9] of every BEL to a *random* value and generate a bitstream. Using a random value is essential as fixed or predictable patterns could hide subtle frame address indexing bugs. Finally, we use our device and architectural parameters to read the bitstream and extract the value used as the startup configuration for each BEL. If we can successfully recreate every initial value, we can assume that the derived parameters are correct. We perform this experiment for both UltraScale and UltraScale+ devices and confirm that all reconstructed configurations match those provided to Vivado. This confirmation, in turn, supports our hypothesis that all tiles of the same type have the same format in the bitstream and are translated versions of one another.

In summary, the techniques described in this chapter can correctly locate and modify specific configuration bits in Xilinx's UltraScale and UltraScale+ FPGA bitstreams. We believe the same techniques will work for other product lines with minor adaptations. For example, Xilinx's 7-series FPGAs have a similar frame address format to that of UltraScale devices as the minor column, major column, and block type fields are identically-placed and sized to their UltraScale counterparts [98]. The only difference is that the major row field in 7-series FPGAs is one bit narrower than UltraScale devices, and that a "top/bottom" bit is used to select between the top and bottom halves of the device's rows. As a result, it is likely the techniques presented in this chapter are directly applicable to 7-series devices. We leave this to future work.

7.8 Discussion

This section presents a more in-depth view into the format of different BELs in the bitstream and relates some aspects of their format to the physical layout of the FPGA. We also discuss additional pitfalls to avoid when attempting to understand the bitstream.

7.8.1 Minor Counts per Major Resource Column

Table 7.3 reports the number of minor columns needed to configure CLB, BRAM, and DSP columns in UltraScale and UltraScale+ FPGAs (using the frame enumeration method described in Section 7.4.2).

[9] For LUTs and registers we set the INIT property. For BRAM contents we set the INIT_xx property. For BRAM parity we set the INITP_xx property. For BRAM output registers we set the INIT_A and INIT_B properties.

Resource	UltraScale		UltraScale+	
	#frames	#words	#frames	#words
CLB	12	1476	16	1488
BRAM	128	15744	256	23808
DSP	4 (6)	492 (738)	8	744

Table 7.3 – Bitstream configuration word breakdown. We report the number of frames required to configure different major resource columns in UltraScale and UltraScale+ FPGAs. We also list the number of 32-bit words needed to configure each major resource. Frames are composed of 123 words in UltraScale devices, and 93 words in UltraScale+ devices [99].

UltraScale+ devices require more minors to configure each resource. However, the frame size in the two architectures is different: frames are composed of 123 words[10] in UltraScale devices, whereas they comprise 93 words in UltraScale+ devices [99]. The number of 32-bit words needed to configure a column of CLBs is roughly equivalent between both architectures. However, the number of words needed to configure BRAM columns and DSP columns is significantly higher in UltraScale+ devices.

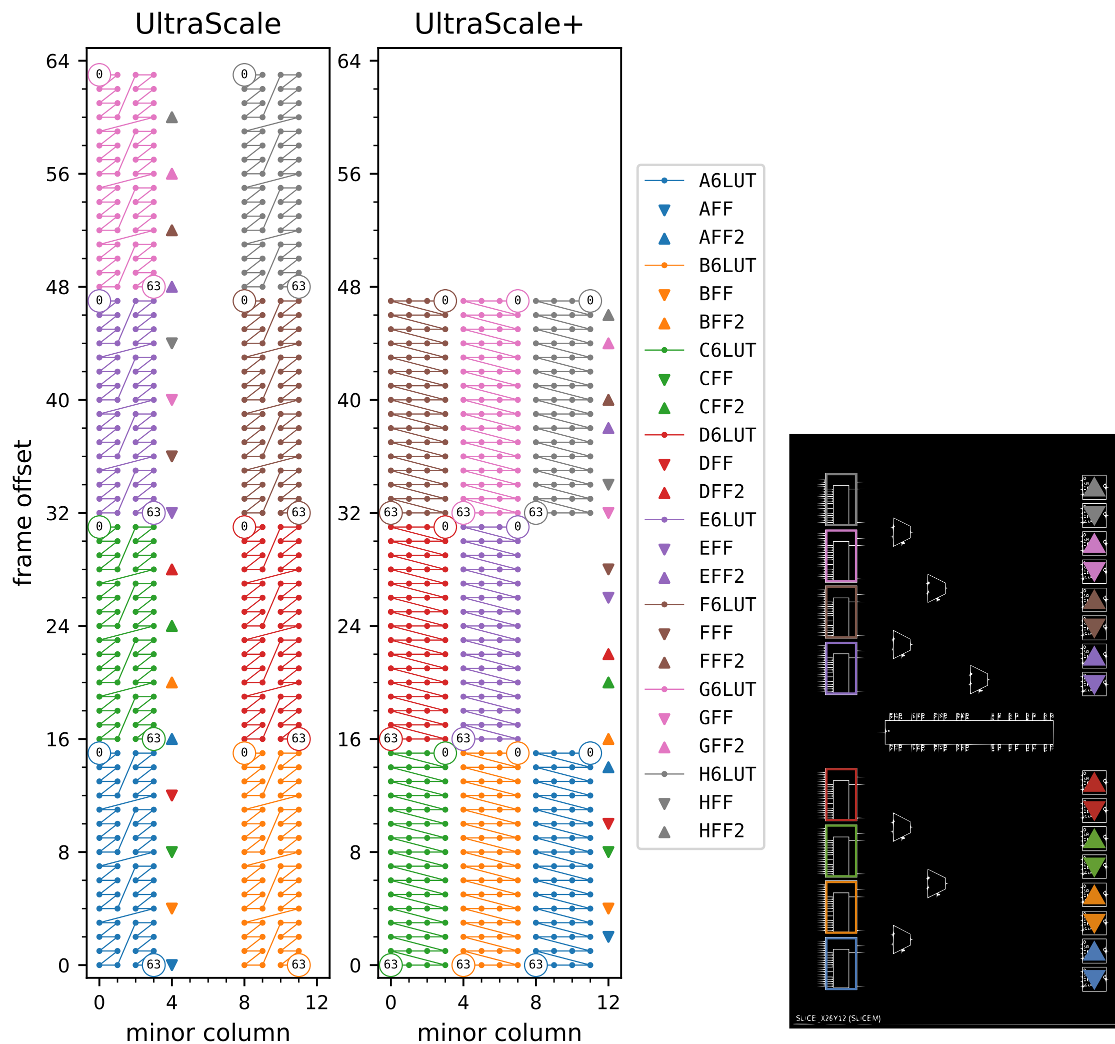
Pitfall 7 explains the peculiar case of DSP columns in UltraScale devices.

Pitfall 7: DSP columns in UltraScale devices have heterogeneous minor counts

A hypothesis for forming the SLR frame indexing scheme in Figure 7.3 was that any two major columns that contain the same resource must have the same number of minor columns, otherwise the resources in the column would not be homogeneous. While this is true in general, there seem to be device-specific deviations.

BitMan [75] reported that DSP major columns in UltraScale devices contain 4 minors. The frame enumeration scheme presented in Section 7.4.2 confirms this number, but not for all DSP columns. We found that DSP columns that are adjacent to a clock region boundary are configured with 6 frames. It is unclear why extra configuration frames would be necessary for these clock region boundary columns. It is possible this requirement is device-specific as only two devices form our UltraScale device lineup (see Section 7.7), one of which is likely a restricted version of the other (see Pitfall 3).

In contrast, UltraScale+ devices use 8 frames to configure DSP columns irrespective of where they are located. This was confirmed on the 38 UltraScale+ devices used in our evaluation.



(a) The architecture-specific minor columns and frame offsets of the BELs in the bottom-most CLB in a clock region. The LUTs each have a 64-bit INIT property. We highlight the first and last bits to identify the order in which the INIT bits are placed. The flip-flops each have a 1-bit INIT property. Each LUT is associated to two flip-flops. The triangles whose tips face downwards correspond to flip-flops [A-H]FF. The triangles whose tips face upwards correspond to flip-flops [A-H]FF2.

(b) Zoomed-in image of a single CLB from the Vivado device viewer. The 8 colored rectangles on the left each represent a LUT. The 16 colored triangles on the right each represent one flip-flop. The legend matches that of the figure to the left.

Figure 7.8 – LUT and flip-flop INIT bits in UltraScale and UltraScale+ devices. The figure lists the INIT bits of tile CLEL_L, but we found the INIT bits of all other CLB tiles to be at identical minors and frame offsets.

7.8.2 CLB Format

Each CLB column in UltraScale and UltraScale+ devices contains 60 CLBs [101]. CLBs in the same column are necessarily of the same tile type and share a common format in the bitstream

[10] A word is a 32-bit value in Xilinx terminology.

(i.e., they are translated versions of each other; see Section 7.6.2). It is therefore sufficient to detail the format of a single CLB.

The left side of Figure 7.8 compares the format (minors and frame offsets) of the 8 LUTs and 16 flip-flops contained in the bottom-most CLB of a clock region in UltraScale and UltraScale+ devices. We highlight each LUT and its two flip-flops with the same color. We mark the first and last bit of each LUT's 64-bit INIT property. Each flip-flop has a 1-bit INIT property and therefore does not require special marking.

The UltraScale and UltraScale+ architectures share the same CLB datasheet [101] and CLB block diagram in the Vivado device viewer (right side of Figure 7.8), yet the left side of Figure 7.8 shows that the configuration bits of LUTs and flip-flops follow different placement patterns.

The number of frames that are needed to configure the LUTs and flip-flops also differ. The LUTs and flip-flops in UltraScale devices are configured by writing to 9 minors (8 minors for the LUTs and 1 minor for the flip-flops) with a total of 1107 words. However, the same BELs in UltraScale+ devices are configured by writing to 13 minors (12 minors for the LUTs and 1 minor for the flip-flops) with a total of 1209 words.

While Table 7.3 shows that UltraScale+ devices use 16 minors for their configuration, our experiments show that only minors 0–12 are used for LUTs and flip-flops. Minors 13–15 are likely used to configure the other BELs in a CLB (e.g., F-MUXes or carry logic), or the routing between CLB columns.

Figure 7.8 shows the format of INIT bits in a CLB tile of type CLEL_L. However, plotting the same figure for all other CLB tiles reveals that they all share the same format for INIT bits. LUTRAM tiles (CLEM and CLEM_R) physically support more functionality that standard LUT tiles (CLEL_L and CLEL_R), yet the extra programmability of LUTRAMs does not change the position of INIT bits. It is likely standard LUT tiles are simply LUTRAM tiles with the extra circuitry removed, but with the same programming circuitry layout.

Finally, flip-flop values are stored in an inverted state in the bitstream at the designated minor and frame offset [95].

7.8.3 BRAM Format

BRAMs are 18Kb in size, with 16K data bits and 2K parity bits. Vivado uses different properties to specify the initial contents of data and parity bits. The 16K data configuration bits are specified in 256-bit properties INIT_00–INIT_3F. The 2K parity configuration bits are specified in additional 256-bit properties INITP_00–INITP_07. To simplify identifying a given bit, we linearize the individual 256-bit INIT_xx properties into single 16K data and 2K parity properties.

Figure 7.9 compares the format of initial configuration bits in the bottom-most BRAM in a clock region in UltraScale and UltraScale+ devices. Linking the bits individually (as in Figure 7.8 for a LUT) does not easily show the stride between each point as the configuration bits increment

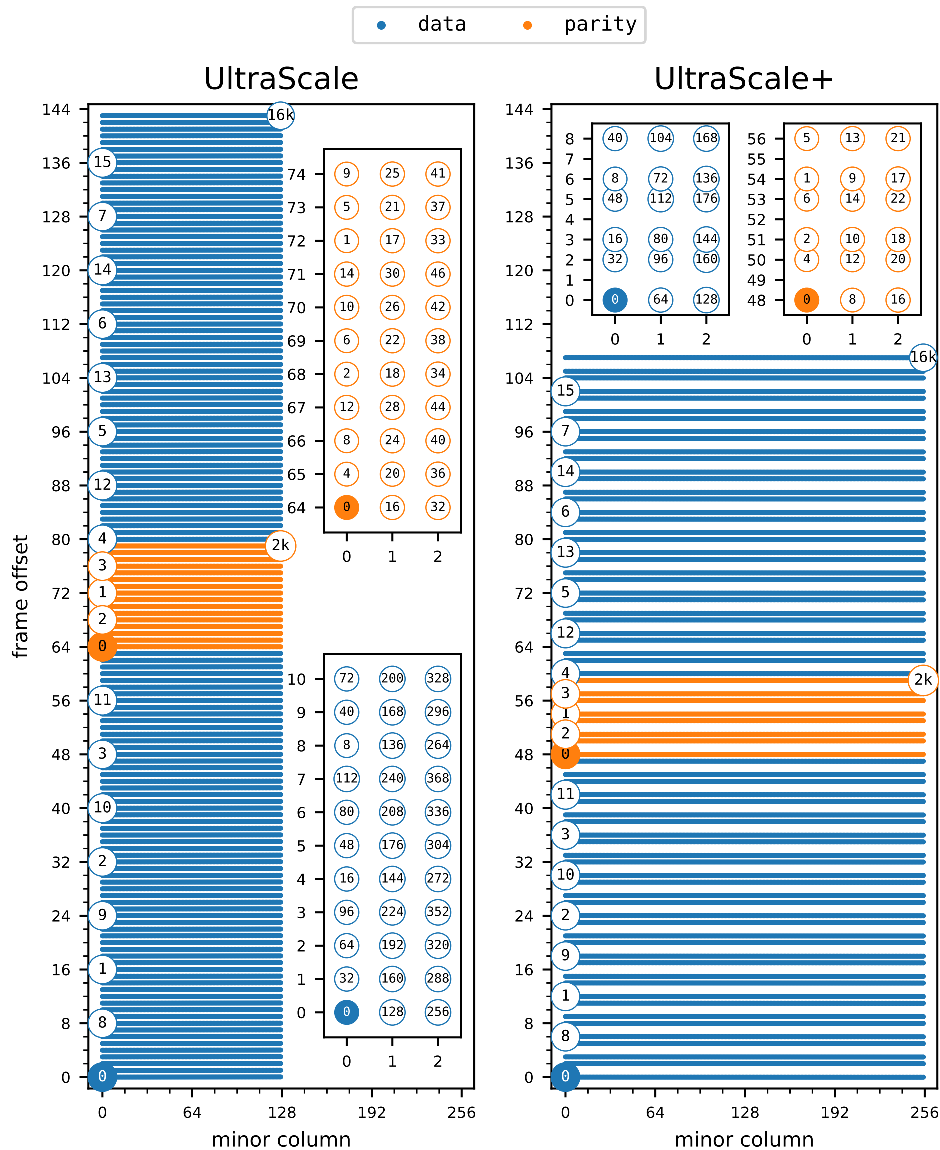


Figure 7.9 – BRAM INIT bits in UltraScale and UltraScale+ devices. The minor columns and frame offsets are reported for the bottom-most BRAM in a clock region. Each BRAM contains 18Kb divided into a 16Kb data array (blue) and 2Kb parity array (orange). The two internal figures shown in each architecture are zoomed-in versions of the larger plot centered around bit 0 of the data and parity bits.

in the same minor before passing to the next one, resulting in a vertical line if linked together. Instead we plot the first 16 data bits and 4 parity bits in a zoomed-out figure to show the high-level stride, then zoom into the region around data bit 0 and parity bit 0 to reveal more details.

Table 7.3 shows that UltraScale devices use 128 minors to configure a column containing 24 18Kb BRAM, whereas UltraScale+ devices use 256 minors to achieve the same. As a result, UltraScale devices use a denser bit pattern compared to UltraScale+ ones, with UltraScale+ devices skipping odd and even frame offsets in an interleaved fashion. Full data tables are available in the Bitfiltrator repository [57].

Each 18Kb BRAM contains two 18-bit hard output registers (not shown), i.e., they are contained within the BRAMs themselves and are not CLB flip-flops. While the output registers are inside BRAMs, their initial contents are specified in the `CLB_IO_CLK` region of the bitstream (i.e., `BRAM_CONTENT` is reserved for BRAM *memory* contents).

7.8.4 Frame Offsets and Physical Placement

We know frames all have the same size and span one clock region vertically (see Section 7.3.1), so we suspect there is a relationship between the vertical physical placement of a BEL and its architecture-specific frame offset.

The left side of Figure 7.10 plots the frame offset of `A6LUT.INIT[0]` across all CLBs in a clock region column[11]. We observe a regular frame offset increment until the 30th CLB, a sharp rise between the 30th and 31st CLBs (the middle two CLBs in a column), and finally a regular increment until the last CLB in a column. Similar plots for the BRAMs show the same gap between the middle two BRAMs of a column.

The right side of Figure 7.10 explains this gap by zooming into the area where the frame offset gap occurs. We observe a physical gap between the middle two tiles of all resource columns. Zooming in further reveals this space is taken up by each column's clocking resources.

We conclude that a BEL's frame offset does correlate with its vertical physical placement in a clock region, and is a viable proxy to locate resources in the device when reverse-engineering. Pitfall 8 cautions that a generalization of this approach cannot be made for minor columns without more information.

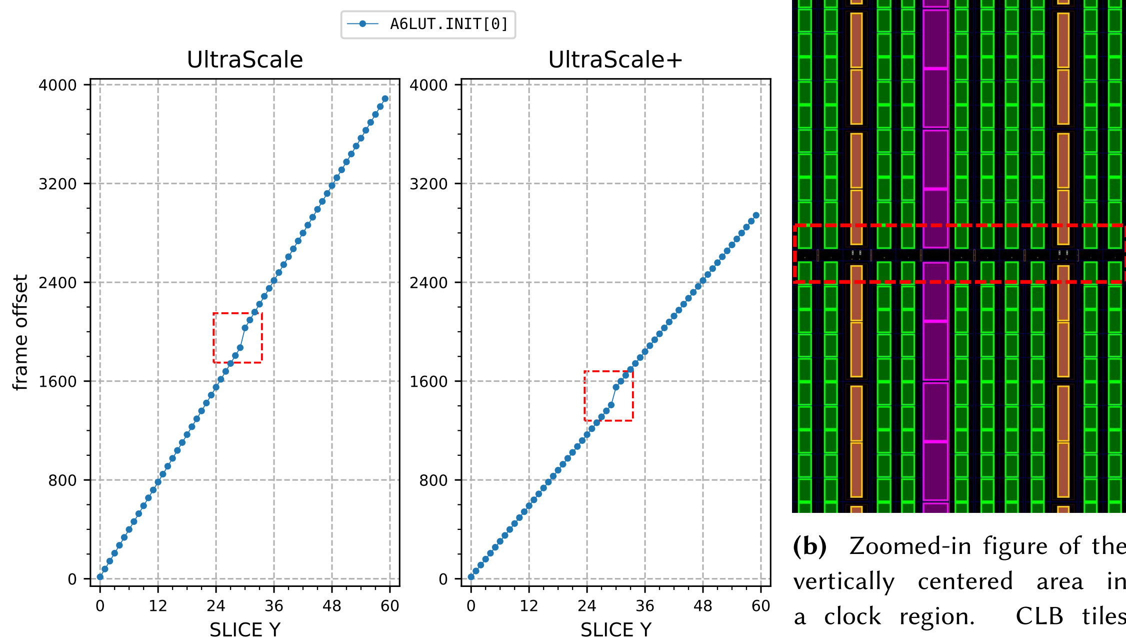
Pitfall 8: Minor columns are not proxies for physical placement

While a configuration bit's frame offset is a proxy for its vertical physical placement in a clock region (see Section 7.8.4), we cannot conclude that the same configuration bit's minor column is a proxy for its horizontal physical placement inside the resource column. Indeed, the left side of Figure 7.8 shows that flip-flops in UltraScale devices are located at minor column 4, i.e., to the *left* of half of the LUTs. However, the right side of the same figure shows that flip-flops are to the *right* of the other BELs in the CLB.

7.8.5 SLR Similarities

One way to determine whether two devices have identical resources is to compare their frame addresses. If all their frame addresses match, then it is likely the two devices share the same column mix. Frame addresses must be compared at the SLR level as each SLR is an independent FPGA and is the smallest unit at which we can differentiate two devices.

[11] We do not need to plot the frame offsets of other BELs in the CLBs as we know all BELs have a fixed relative position from each other.



(a) Frame offset of A6LUT.INIT[0] for the 60 CLBs in a resource column. The jump in frame offsets between the 30th and 31st CLBs matches the physical gap between the CLBs taken up by clocking resources. BELs in other resources exhibit similar frame offset jumps at the same location.

(b) Zoomed-in figure of the vertically centered area in a clock region. CLB tiles (green), BRAM tiles (orange), and DSP tiles (magenta) are all densely packed around a central area that contains clocking resources (red) for each column.

Figure 7.10 – Relationship between BEL frame offsets and vertical placement in a clock region.

Table 7.4 groups the 40 devices from our evaluation (see Section 7.7) by their SLRs' frame addresses. Any two devices in the same row are indistinguishable at the binary level. Pitfall 9 then cautions against making quick physical conclusions if two SLRs share the same frame addresses.

Pitfall 9: SLRs with common frame addresses may be physically different

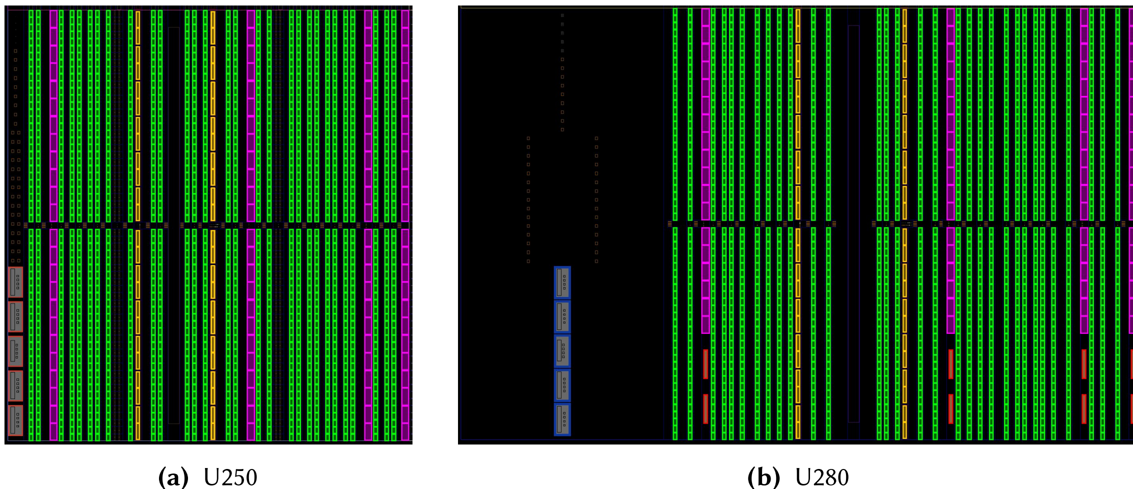
Surprisingly, many devices that are indistinguishable at the binary level may look physically different.

For example, Table 7.4 shows that the xcu250 and xcu280 share the same frame addresses. Both devices do indeed share the same resource columns, including the order in which they appear. However, Figure 7.11 shows that clock region X0Y0 in the Vivado device viewer is physically narrower in the xcu250. The horizontal spacing between resource columns also differ. Finally, we see that DSP columns in the xcu250 are full-height, whereas those in the xcu280 are truncated at the bottom of the clock region, with the remaining space taken up by HBM-related tiles.

The *horizontal* placement of resource columns in a device is therefore independent from its bitstream. This further emphasizes the conclusion from Pitfall 8.

Family	Type	Devices
UltraScale	FPGA	xcku025, xcku035
UltraScale+	SoC	xazu1eg, xczu1cg, xczu1eg
		xazu2eg, xazu3eg, xczu2cg, xczu2eg, xczu3cg, xczu3eg
		xazu4ev, xazu5ev, xck26, xczu4cg, xczu4eg, xczu4ev, xczu5cg, xczu5eg, xczu5ev
		xazu7ev, xcu30, xczu7cg, xczu7eg, xczu7ev
	FPGA	xcau10p, xcau15p
		xcau20p, xcau25p, xcku3p, xcku5p
		xcu25
		xcu50, xcu55c, xcu55n, xcu250, xcu280
		xcu26, xcux35
		xcu200

Table 7.4 – Devices grouped by SLR frame addresses. Any two devices in the same row have the same frame addresses and are indistinguishable at the binary level.

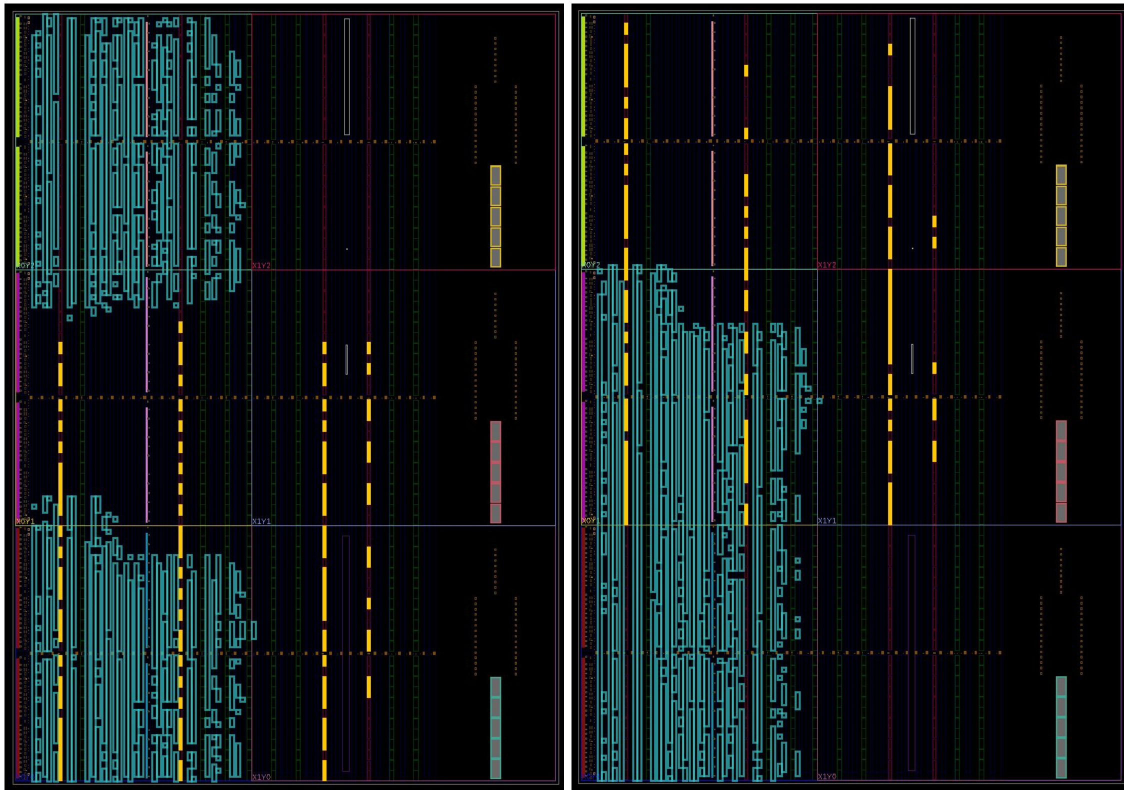


(a) U250 **(b)** U280

Figure 7.11 – Clock region X0Y0 in the Alveo U250 and U280 datacenter-grade FPGAs. Each SLR in the U250 and U280 are indistinguishable at the binary level, yet the two devices have a different physical layout. We highlight CLBs in green, BRAMs in orange, and DSPs in magenta. DSP columns in the U280 are shorter than on the U250 and HBM tiles (in red) take up the remaining space. Finally, the horizontal spacing between resource columns is not the same between devices.

7.8.6 Device Capacity

The frame addresses in a bitstream revealed that many devices share similar SLRs, yet it is interesting to note that Vivado reports different resource counts for them.



(a) Placing 100 BRAMs in the bottom region of the device.

(b) Placing 100 BRAMs in the top region of the device.

Figure 7.12 – Floorplan of a xcau10p device. We highlight BRAMs in orange and all other resources in teal. Vivado reports that this device contains 100 BRAMs, but the device physically contains 288 BRAMs. All BRAMs are functional as placing BRAMs at non-overlapping regions of the device both result in a generated bitstream.

For example, Table 7.4 shows that the xcau10p and xcau15p devices have identical SLRs, but Vivado reports that the xcau10p contains 100 BRAMs and that the xcau15p contains 144 BRAMs. However, counting the number of BRAMs physically available in each device reveals there are, in fact, 288 BRAMs in each! One can craft different designs for the xcau10p where all 100 BRAMs are used in non-overlapping positions (see Figure 7.12). The design is legal and a bitstream is generated, which shows that *all* 288 BRAM are functional. Similar observations hold for the remaining resources in the device. Even the largest datacenter-grade FPGA, the Alveo U250, exposes fewer BRAM resources in software than are available on the device.

The bitstream format therefore reveals that more resources are available than those shown in Vivado. The resource limits are likely just a software restriction.

7.9 Summary

Bitstream manipulation is a necessary step to use an FPGA in novel ways, but is currently unsupported by vendor tools. The undocumented FPGA bitstream file format requires reverse

engineering to find the information to support this step. We detailed a systematic, top-down approach to reverse-engineer the format and location of specific cells in Xilinx FPGAs. We also link various observations of a bitstream's properties to physical aspects of the device it programs. To the best of our knowledge, this is the first work to *explain* the foundational techniques for bitstream reverse-engineering. We believe these techniques will be valuable in reverse engineering other FPGAs and improve open FPGA design flows.

Part III Related and Future Work

8 Related Work

8.1 Software Simulators, FPGA Prototypes, and Emulators

There are multiple ways to perform functional verification depending on one's constraints and goals.

Simulation consists of running a hardware design in *software*, typically on a single machine. Software simulators excel at hardware debug as they provide excellent visibility into design state. They also allow HW/SW integration testing in small- to medium-sized designs. However, software simulators are generally too slow to perform software validation. Many open-source and commercial software RTL simulators exist, each with various design choices and corresponding simulation rates: iVerilog, Verilator, QuestaSim, VCS, etc.

Prototyping implies running a design on *programmable hardware*, generally in the form of an FPGA. Prototypes are very fast and achieve interactive simulation speeds by mapping RTL circuits directly into *gates* on one or more FPGAs [114]. Prototypes provide the necessary speed for software validation as they can run full software stacks on top of a simulated circuit over trillions of clock cycles. However, FPGA prototypes are long to compile for as they require placing and routing a design. Prototypes also offer only limited visibility into a design's state, and so are unsuitable for hardware debug or HW/SW integration. A single FPGA is often not large enough to prototype an ASIC, so multiple devices are often needed. MIT's Virtual Wires [8] project attempts to streamline partitioning a single RTL design over multiple FPGAs by virtualizing FPGA pins through multi-pumping. Similarly, modern industrial solutions such as Cadence's Protium X2 support mapping $\approx$ 2.4B gates to 60 FPGAs [34]. FireSim [54] is an open-source FPGA prototyping platform, widely used as an *architectural* simulator for exploring RISC-V designs at datacenter-scale using cloud FPGAs.

Emulation platforms are RTL simulators for very large designs. They greatly increase simulation capacity by running across racks of custom processors or commercial/custom FPGAs [81, 114]. Emulators represent the peak of what the EDA industry provides for functional verification and are suitable for all levels of the design cycle (hardware debug, HW/SW integration, and software validation). They also provide rich debugging environments that are designed to find design

bugs in the hardware or software of a complex system (e.g., in a GPU [19]). Emulators remain functional verification tools, so FPGA-based implementations typically omit various aspects of synthesis related to physical optimization to enable faster compilation [81]. Though pricing for large EDA companies' emulators are not public, estimates from industry experts hover around $1M+ [47, 80].

Although Manticore is implemented on an FPGA, its simulation runs in software (a program running on Manticore) rather than being mapped to an FPGA. Consequently, Manticore's compile times are a few minutes, whereas FPGA prototypes take hours to days to compile. Manticore's design is much closer to that of an emulator and is a first step towards an open-source alternative to commercial emulation platforms.

8.2 Custom Functions

Manticore is not the first work to propose using custom functions in RTL simulation. IBM's Yorktown Simulation Engine (YSE) [36] did something similar 40 years ago! YSE implements special purpose *logic processors* capable of computing arbitrary 2-bit wide logic functions using table lookups.

More recently, Nexus [13] proposed an FPGA-accelerated emulation platform for small designs using a mesh of statically-scheduled processors linked with a dynamically-scheduled NoC. Nexus' processors support only truth tables.

Manticore uses 4-input functions to compress bitwise-parallel logic operations. We found custom functions cannot provide more than $\approx 10\%$ improvement in simulation speed with its current implementation. Relaxing the constraint that a function's inputs must be bitwise-parallel will likely improve performance, but at the cost of increased physical implementation complexity in each core.

8.3 Sequential RTL Simulation

Most efforts in improving RTL simulation on CPUs focused on reducing the runtime overhead of event-driven simulation.

ESSENT [10, 11] is a hybrid cycle-accurate simulator that employs a coarsened, conditional, singular, static (CCSS) execution model. CCSS is a novel, hybrid approach that minimizes the overhead of runtime checks in event-driven simulation, especially in the presence of low activity factors. ESSENT is single-threaded and accelerates simulation of RISC-V cores (CPUs have low activity factors) by 1.5–11.5× over Verilator. However, it is not clear how ESSENT performs with spatial designs that exhibit high activity factors. Manticore's performance is independent of a design's activity factor.

Cuttlesim [76] is a cycle-accurate simulator for Kôika [14], a rule-based HDL derived from Bluespec Verilog [68]. Cuttlesim uses the high-level semantics of Kôika to generate C++ code

optimized for sequential performance. It reports 2–3× faster simulation than the equivalent RTL code running serial Verilator.

8.4 Parallel RTL Simulation Using CPUs

RepCut [110] is a full-cycle, cycle-accurate RTL simulator that targets execution on shared-memory machines. RepCut and Manticore independently and simultaneously proposed the same idea: using a BSP simulation algorithm to batch synchronization and inter-thread communication between cores to a single point. Like Manticore, RepCut also uses replication-aided partitioning to split RTL simulation between multiple cores. RepCut reports a superlinear parallel speedup of ≈27× using 24 threads compared to single-threaded Verilator. This superlinear speedup is explained by the code footprint of an RTL model, which is far too large to fit in a single core's instruction cache, leading to frequent cache misses and low IPC. Relieving cache pressure by partitioning the design significantly increases cache locality and IPC, hence per-thread performance increases. Coupled with actual parallel execution, this leads to a superlinear speedup. Manticore is insensitive to cache effects by design as it contains only deterministic hardware datapaths. The main difference between RepCut and Manticore is their scaling properties: our model of parallel simulation on a shared-memory machine (see Section 3.4.3) showed that the simulation rate drops past few tens of cores, which both our experiments and RepCut's results confirm. A shared-memory machine can achieve only weak scaling as more cores can be effectively used only by increasing the workload size. By contrast, Manticore's architecture removes the cost of synchronization and achieves strong scaling.

Metro-MPI [64] is a cycle-accurate RTL simulator for very large SoC designs (e.g., 10B transistors). Its key insight is to partition a modern SoC along its natural boundaries (e.g., NoCs) to turn RTL simulation into a distributed high-performance computing (HPC) problem. It reports speedups of 136× against serial Verilator and 9.3× against multithreaded Verilator when simulating a 1024-core OpenPiton SoC across 22 machines. Metro-MPI requires that developers manually partition the design and replace its boundary interfaces with MPI calls. It can be applied to arbitrary hardware designs, provided that its interfaces are understood, and so requires expert knowledge of the design to enable high-speed simulation. Metro-MPI is based on distributed execution on shared-memory machines and can achieve only weak scaling; Manticore achieves strong scaling.

DyVe [93] is an event-based, cycle-accurate RTL simulator running on a custom array of many-core SoCs linked with a central FPGA. DyVe partitions the circuit graph by its primary outputs, then incrementally merges program regions that share the largest number of inputs. DyVe's performance numbers are based on whether a target's simulation code fits its processors' L1, L2, or SDRAM memories, so direct comparisons are impossible.

8.5 Parallel RTL Simulation Using GPUs

There is considerable research on accelerating RTL simulation using GPUs. Most of this work focused on reducing the runtime overhead of monitoring value changes and demonstrate significant speedups relative to commercial *event-driven* simulators. By contrast, Manticore is a *full-cycle* simulator, so it is not comparable to these systems. We nevertheless provide a short survey of the most relevant (and recent) work for completeness.

GCS [25–27] is a hybrid GPU-accelerated event-driven simulator that levelizes logic gates in coarser macro-gates for decreased runtime overhead associated with monitoring nets. It reports orders of magnitude faster runtime (5 kHz simulation rate) compared to single-threaded commercial simulators.

Qian and Deng [78] propose a GPU-accelerated event-driven RTL simulation based on the Chandy-Misra-Bryant [15, 23] distributed simulation algorithm. They show up to 50× faster simulation versus a single-threaded commercial simulator and report simulation rates of 37 kHz. However, their evaluation compares against a low-end desktop CPU.

SCGPSim [67] accelerates SystemC [48] simulation on GPUs. SystemC is a C++ library for event-driven simulation of circuit models which conventionally relies on userspace cooperative threads (mapped to a single kernel thread) for modeling concurrent processes. They propose a new thread model for GPU execution and report up to 100× faster simulation time compared to a *laptop CPU* with microbenchmarks.

RTLFlow [63] is a GPU-accelerated cycle-accurate RTL simulator that exploits stimulus-level parallelism to speed up simulation by running many independent simulations on a GPU. RTLFlow improves execution speed by up to 40× over Verilator for many stimuli, but it runs an order magnitude slower than Verilator with a single stimulus. Manticore is faster than Verilator with a single stimulus.

Zhang, Ren, and Khailany [116] called for a renewal in GPU-accelerated RTL simulation research by leveraging recent advances in GPU-compute APIs designed for machine learning.

8.6 Deterministic Acceleration

Manticore's design philosophy is similar to VLIW processors and other Raw machines [108]. A more recent example is the Groq's tensor streaming processor (TSP), a machine learning accelerator [1, 2]. Like Manticore, the TSP has deterministic hardware datapaths that enable precise reasoning and control by software. The TSP achieves determinism by eliminating all reactive elements in its design (e.g., arbiters, caches, etc.) [1, 2]. Like RTL simulation, machine learning exhibits rare long-lived divergent code paths, which makes static scheduling feasible.

9 Future Work

Manticore explores providing architectural support to accelerate RTL simulation by using fine-grain parallelism. This work focused on the technical aspects of our approach, which is still a prototype, not a complete "tool". Nevertheless, our results show a clear performance advantage of the Manticore prototype over a highly optimized software simulator for many examples. At this level of maturity, Manticore is not a replacement for Verilator or other simulators. Much work is needed to bring Manticore to the same level of usability as Verilator, which has enjoyed more than a decade of active development.

We outline short-term engineering improvements and longer-term architectural improvements that would greatly improve Manticore's usability and scalability.

9.1 Language Support

Manticore does not support most of SystemVerilog. Some language features can be solved with a more sophisticated frontend (Yosys' support for SystemVerilog is incomplete). Advanced language features (e.g., event control) are necessary for a complete simulator, especially for writing complicated test benches. However, language features for accurate timing control (i.e., not cycle-accurate) are incompatible with our static scheduling approach and would be challenging to retrofit.

9.2 Multiple Clock Domains

Large RTL designs generally contain multiple clock domains, which Manticore's current implementation does not support: each core automatically jumps back to the start of its instruction memory at the end of compiler-scheduled synchronization and executes instructions unconditionally.

Adding support for multiple clock domains entails compiling the logic driven by each clock separately and placing them in distinct memory regions, which can easily be performed by multiple invocations of the Manticore compiler. Cores must then choose which memory region to

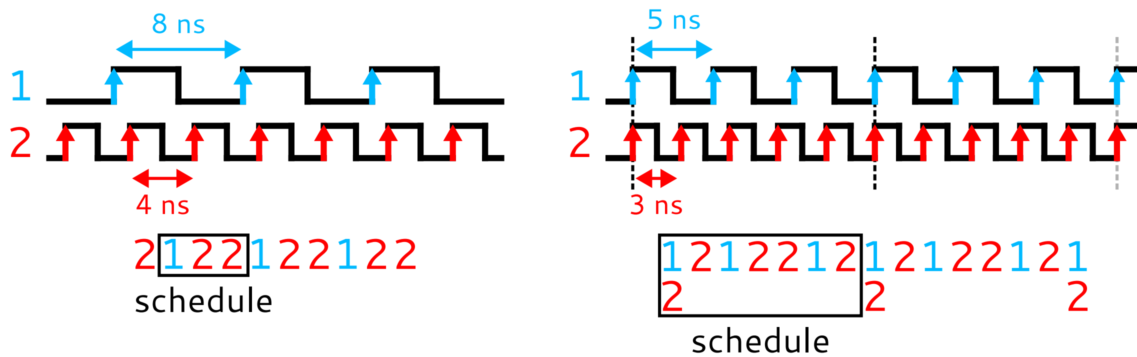


Figure 9.1 – Scheduling clock activation in a full-cycle simulator. We show two cases where logic driven by multiple clock domains can be statically scheduled by a full-cycle simulator. When different clocks are perfect multiples of each other (left), we invoke the logic driven by the slow clock, then follow with multiple invocations of the logic driven by the fast clock. When clocks are not perfect multiples of each other (right), then we invoke both clocks multiple times in a pre-determined pattern, after which the clocks will forcefully align (dashed lines) and the pattern repeats (if the clocks have only integer periods).

jump to at runtime, but Manticore does not support branches. The jump can instead be handled with an architectural change to each core's controller. A simple state machine in each core can choose which clock to activate at each cycle.

If the clocks have a fixed phase relationship—informally, the clock edges do not "drift" eternally—then we can encode a static schedule for toggling the clocks (see Figure 9.1). All clocks with an integer period satisfy this property, but clocks with irrational periods (e.g., $3.\overline{33}$ MHz) do not. If the clocks do not have a fixed phase relationship, then the state machine would need to look up the next times at which each of the clocks toggle and dynamically select the closest one.

9.3 Waveform Debugging

Digital hardware is fully parallel, and so is debugged using *waveforms*: a cycle-by-cycle record of all signals in an RTL design. Waveform debugging is an essential tool in a digital designer's arsenal, which Manticore does not yet support.

It is possible to use Manticore's global store instructions to record values, but it would incur excessive overhead. More realistically we need dedicated hardware support for out-of-band waveform collection. One way is to assign some cores entirely to waveform collection and for other cores to send register updates to them. These cores could then buffer updates and DMA them to off-chip DRAM. These special cores would need to be driven by the control clock and designed with care as they may need to temporarily halt the compute clock if their internal buffers fill up.

9.4 Physical Implementation

Early in the project, we decided to build an FPGA prototype since fabricating an ASIC was not affordable, and simply simulating a Manticore processor would yield less information than constructing an implementation. The FPGA implementation, however, is limited in its clock frequency, number of cores, and simulation capacity.

Manticore's current design is clocked at 475 MHz, which is near the realistic maximum clock frequency of 500 MHz to 600 MHz that an FPGA can achieve. Further increasing the clock frequency is unlikely and the only way to do so would be with an ASIC implementation, which has its own challenges at high clock frequencies. More realistically the improved delays made possible by an ASIC implementation should be used to significantly decrease Manticore's pipeline depth, which is excessively long due to reach ill-placed FPGA resources (URAMs). In any case, a detailed study of the tradeoff between pipeline depth, clock frequency, and their impact on partitioning imbalance between cores (due to excessive `NOP`s needed to handle data hazards) is needed in future core designs.

Manticore's core count is limited by the FPGA's scarce URAM resources, which are used for dense instruction memories. Our prototype can simulate up to ≈ 900k instructions (4096 instructions in each of 225 cores) with about 14.4 MiB SRAM for data and instruction. ASICs can easily provision hundreds of MiB of SRAM [2, 50].

We can increase simulation capacity by (1) shrinking instruction widths, and (2) decreasing the number of instructions needed to emulate an RTL design. We can easily divide the instruction width by two[1] so we can put more instructions in a given instruction memory, but this just allows doing more work in each core and does not enable more parallelism. We could further increase simulation capacity by reducing the number of instructions needed to emulate an RTL design. One avenue is to consider a wider 32-bit datapath (vs. Manticore's current 16-bit datapath) as most accelerators have 32-bit interfaces, and so would require fewer instructions—on average—to simulate. An alternative to increase simulation capacity on FPGAs is to use a device with more URAMs. Unfortunately larger FPGAs do not contain significantly more URAMs, and so increasing the core count on a single FPGA is unlikely. Multiple devices are needed to scale simulation to larger RTL designs.

9.5 Multi-FPGA Simulation

Pushing Manticore's strong scaling beyond a single FPGA is not trivial, but is not impossible either. The main challenge in scaling out Manticore's design resides in its static schedule: while it is simple to maintain determinism in a single system, it is unclear how to do so in a distributed system as clocks do not originate from the same source. This is further compounded by the fact that distinct FPGAs can "drift" from one another as dynamic events like off-chip DRAM accesses cause dynamic clock gating in Manticore's design.

[1] We can replace Manticore's large 2W4R register files with smaller 1W2R register files.

One solution would be to add a global counter to each FPGA and to augment Manticore with logic to ensure counters on different devices stay aligned to within a certain interval, which the compiler then uses to correctly schedule code on all devices. This can be performed by continually exchanging global counters between pairs of FPGAs and periodically clock-gating devices until they converge back to a the target interval. Similar hardware-aligned counters are used by Groq's TSP chip to link multiple devices together within and across a rack [1].

9.6 Timing-Accurate Simulation

In principle timing-accurate simulation is unnecessary if designs are constrained correctly and static timing analysis (STA) is successful. However, informal discussions with engineers at a chip vendor revealed that timing-accurate simulation is nonetheless always used to ensure correctness; the economic impact of taping-out a potentially buggy chip due to invalid STA is far too high to skip simulation.

Unfortunately timing-accurate simulators are orders of magnitude slower than cycle-accurate simulators and run at just a few Hz for large designs—not kHz! Furthermore, timing-accurate simulation is performed late in the design process and extensively before tape-out, a period when design teams are already under extreme time pressure. Speeding up timing-accurate simulation would likely provide a significant impact to all players in the chip industry.

An open question is whether timing-accurate simulation can be parallelized effectively and whether Manticore-like architectures would be suitable platforms to do so.

9.7 Using Accelerators Designed for Other Problem Domains

Considering the economic constraints of chip design, designing dedicated chips will continue to be economically viable for only high-volume applications. Low-volume applications will always be relegated to FPGA-based implementations, which—while powerful—will always lag behind ASICs. The question is whether we can use silicon investments in other problem domains to accelerate RTL simulation?

RTL simulation is not the only workload that benefits from having access to a large amount of SRAM. Given the massive commercial interest in machine learning, the computing industry has poured a huge amount of money into designing hardware accelerators to apply machine learning at scale. Most machine learning accelerators appear to be promising platforms for research in accelerated RTL simulation: they (1) contain hundreds of MiB of SRAM to store the abundant weights needed to run machine learning models, (2) are often manycore MIMD designs, and (3) contain efficient interconnects for communication within and beyond a single chip. Some of these devices also support constant-time synchronization, the key design trait needed to enable fast RTL simulation. Future research should evaluate whether these massively-parallel architectures can be repurposed for scalable RTL simulation.

10 Conclusion

The demise of Moore's Law and Dennard scaling has resulted in diminishing performance gains for general-purpose processors, and so has prompted a surge in academic and commercial interest for hardware accelerators. Specialized hardware has already redefined the computing landscape by enabling the emergence of disruptive, large-scale applications that would otherwise not have been possible with CPUs alone.

RTL simulators play a key role in enabling the accelerated computing revolution: they are to hardware engineers what debuggers and runtime systems are to software engineers. Without RTL simulators, no hardware accelerator could be functionally designed, let alone its performance validated. As accelerators increase in size and complexity, the hardware design industry will increasingly need faster RTL simulators to permit chip design in tractable time frames.

Parallelism is the preferred approach to improve software performance. This thesis argues that systems for RTL simulation should be designed with architectures that permit strong parallel performance scaling—which permits effective use of increased parallelism without the need to increase workload sizes—to reduce simulation turnaround time.

This thesis contributes ideas and techniques that led to the design and implementation of Manticore: a strong scaling manycore architecture purpose-built for accelerated RTL simulation. Manticore combines a bulk-synchronous parallel execution model with static scheduling to eliminate the runtime overheads of synchronization among hundreds of cores, simplify core design, and significantly increase the parallelism possible on a single chip. Manticore's modest 225-core FPGA prototype consistently achieves better performance than a state-of-the-art software RTL simulator running on top-of-the-line desktop and server x86 processors. The Manticore system demonstrates the actual performance benefits of exploiting fine-grained parallelism in RTL code to accelerate simulation. Its higher speed allows several long simulations per day, as opposed to several per week on a conventional computer, leading to a clear improvement in developer productivity. Manticore presents a first step towards fast, open-source, scale-out RTL simulation.

Bibliography

[1] Dennis Abts, Garrin Kimmell, Andrew C. Ling, John Kim, Matthew Boyd, Andrew Bitar, Sahil Parmar, Ibrahim Ahmed, Roberto DiCecco, David Han, John Thompson, Michael Bye, Jennifer Hwang, Jeremy Fowers, Peter Lillian, Ashwin Murthy, Elyas Mehtabuddin, Chetan Tekur, Thomas Sohmers, Kris Kang, Stephen Maresh, and Jonathan Ross. "A Software-Defined Tensor Streaming Multiprocessor for Large-Scale Machine Learning". In: *Proceedings of the 49th International Symposium on Computer Architecture*. New York, NY, USA, June 2022, pp. 567–580. DOI: 10.1145/3470496.3527405 (cit. on pp. 140, 144).

[2] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E. R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. "Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads". In: *Proceedings of the 47th International Symposium on Computer Architecture*. Valencia, Spain, June 2020, pp. 145–158. DOI: 10.1109/ISCA45697.2020.00023 (cit. on pp. 140, 143).

[3] *Alveo MA35D*. https://www.xilinx.com/applications/data-center/video-imaging/alveo-ma35d.html. Retrieved May 2023. Xilinx (cit. on p. 3).

[4] Tim Ansell. *Project U-Ray: Xilinx UltraScale Bitstream Documentation (pre-built databases)*. https://github.com/f4pga/prjuray-db/tree/master/zynqusp. Retrieved Mar 2022. July 2020 (cit. on p. 124).

[5] Sameh Attia and Vaughn Betz. "StateMover: Combining Simulation and Hardware Execution for Efficient FPGA Debugging". In: *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Seaside, CA, USA, Feb. 2020, pp. 175–185. DOI: 10.1145/3373087.3375307 (cit. on p. 108).

[6] *AWS Inferentia*. https://aws.amazon.com/machine-learning/inferentia/. Retrieved May 2023. Amazon (cit. on p. 3).

[7] *Azure Pricing Calculator*. https://azure.microsoft.com/en-us/pricing/calculator/. Retrieved Oct 2022. Microsoft (cit. on p. 99).

[8] Jonathan Babb, Russell Tessier, Matthew Dahl, Silvina Hanono, David M. Hoki, and Anant Agarwal. "Logic Emulation with Virtual Wires". In: *IEEE Transactions on Computer-Aided*

Design of Integrated Circuits and Systems 16.6 (June 1997), pp. 609–626. DOI: 10.1109/43.640619 (cit. on p. 137).

[9] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. "Chisel: Constructing Hardware in a Scala Embedded Language". In: *Proceedings of the 49th Design Automation Conference*. San Francisco, California, June 2012, pp. 1216–1225. DOI: 10.1145/2228360.2228584 (cit. on p. 48).

[10] Scott Beamer. "A Case for Accelerating Software RTL Simulation". In: *IEEE Micro* 40.4 (May 2020), pp. 112–119. DOI: 10.1109/MM.2020.2997639 (cit. on p. 138).

[11] Scott Beamer and David Donofrio. "Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation". In: *Proceedings of the 57th Design Automation Conference*. San Francisco, CA, USA, July 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218632 (cit. on pp. 15, 138).

[12] René Beuchat, Florian Depraz, Andrea Guerrieri, and Sahand Kashani. *Fundamentals of System-on-Chip Design on Arm Cortex-M Microcontrollers*. Cambridge, UK: Arm Education Media, 2021. ISBN: 9781911531340 (cit. on p. iv).

[13] Peter Birch. "Open Source FPGA-Based Emulation with Nexus". Workshop on Open-Source EDA Technology (WOSET). Nov. 3, 2022. URL: https://woset-workshop.github.io/PDFs/2022/1-Birch-paper.pdf (cit. on p. 138).

[14] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. "The Essence of Bluespec: A Core Language for Rule-Based Hardware Design". In: *Proceedings of the 41st International Conference on Programming Language Design and Implementation*. London, UK, June 2020, pp. 243–257. DOI: 10.1145/3385412.3385965 (cit. on p. 138).

[15] R. E. Bryant. *Simulation of Packet Communication Architecture Computer Systems*. Tech. rep. Boston, MA, USA: MIT, 1977 (cit. on p. 140).

[16] Thang Nguyen Bui and Curt Jones. "Finding Good Approximate Vertex and Edge Partitions is NP-Hard". In: *Information Processing Letters* 42.3 (May 1992), pp. 153–159. DOI: 10.1016/0020-0190(92)90140-Q (cit. on p. 65).

[17] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. "Recent Advances in Graph Partitioning". In: *Algorithm Engineering: Selected Results and Surveys*. Springer International Publishing, 2016. ISBN: 9783319494876 (cit. on p. 98).

[18] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Michael Dahlin, Lizy Kurian John, Calvin Lin, Charles R. Moore, James H. Burrill, Robert G. McDonald, and William Yode. "Scaling to the End of Silicon with EDGE Architectures". In: *IEEE Computer* 37.7 (July 2004), pp. 44–55. DOI: 10.1109/MC.2004.65 (cit. on p. 65).

[19] Nate Burr. *Emulating NVIDIA GPUs*. https://www.youtube.com/watch?v=650yVg9smfI. Apr. 2016 (cit. on p. 138).

[20] Lukai Cai and Daniel Gajski. "Transaction Level Modeling: An Overview". In: *Proceedings of the 1st International Conference on Hardware/Software Codesign and System Synthesis*. Newport Beach, CA, USA, Oct. 2003, pp. 19–24. DOI: 10.1145/944645.944651 (cit. on p. 12).

[21] Lorenzo De Carli, Yi Pan, Amit Kumar, Cristian Estan, and Karthikeyan Sankaralingam. "PLUG: Flexible Lookup Modules for Rapid Deployment of New Protocols in High-Speed Routers". In: *Proceedings of the ACM SIGCOMM 2009 Conference*. Barcelona, Spain, Aug. 2009, pp. 207–218. DOI: 10.1145/1592568.1592593 (cit. on p. 65).

[22] Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. "More Recent Advances in (Hyper)Graph Partitioning". In: *ACM Computing Surveys* 55.12 (Mar. 2022), 253:1–253:38. DOI: 10.1145/3571808 (cit. on p. 98).

[23] K.M Chandy, Victor Holmes, and J. Misra. "Distributed Simulation of Networks". In: *Computer Networks (1976)* 3.2 (Apr. 1979), pp. 105–113. DOI: https://doi.org/10.1016/0376-5075(79)90009-6 (cit. on p. 140).

[24] Colin C. Charlton, D. Jackson, and Paul H. Leng. "Lazy Simulation of Digital Logic". In: *Computer-Aided Design* 23.7 (Sept. 1991), pp. 506–513. DOI: 10.1016/0010-4485(91)90049-3 (cit. on p. 14).

[25] Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco. "Event-Driven Gate-Level Simulation with GP-GPUs". In: *Proceedings of the 46th Design Automation Conference*. San Francisco, CA, USA, July 2009, pp. 557–562. DOI: 10.1145/1629911.1630056 (cit. on p. 140).

[26] Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco. "Gate-Level Simulation with GPU Computing". In: *ACM Transactions on Design Automation of Electronic Systems* 16.3 (June 2011), 30:1–30:26. DOI: 10.1145/1970353.1970363 (cit. on p. 140).

[27] Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco. "GCS: High-Performance Gate-Level Simulation with GPGPUs". In: *Proceedings of the 12th Conference on Design, Automation and Test in Europe*. Nice, France, Apr. 2009, pp. 1332–1337. DOI: 10.1109/DATE.2009.5090871 (cit. on p. 140).

[28] Der-San Chen, Robert G. Batson, and Yu Dang. *Applied Integer Programming: Modeling and Solution*. Hoboken, NJ, USA: John Wiley & Sons, 2010. ISBN: 9780470373064 (cit. on pp. 71, 78).

[29] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks". In: *Proceedings of the 43rd International Symposium on Computer Architecture*. Seoul, South Korea, June 2016, pp. 367–379. DOI: 10.1109/ISCA.2016.40 (cit. on p. 3).

[30] Tim Coe, Terje Mathisen, Cleve Moler, and Vaughan Pratt. "Computational Aspects of the Pentium Affair". In: *IEEE Computational Science and Engineering* 2.1 (1995), pp. 18–30. DOI: 10.1109/99.372929 (cit. on p. 4).

[31] Jason Cong and Yuzheng Ding. "Combinational Logic Synthesis for LUT Based Field Programmable Gate Arrays". In: *ACM Transactions on Design Automation of Electronic Systems* 1.2 (Apr. 1996), pp. 145–204. DOI: 10.1145/233539.233540 (cit. on p. 76).

[32] Jason Cong, Peng Li, Bingjun Xiao, and Peng Zhang. "An Optimal Microarchitecture for Stencil Computation Acceleration Based on Non-Uniform Partitioning of Data Reuse Buffers". In: *Proceedings of the 51st Design Automation Conference*. San Francisco, CA, USA, June 2014, 77:1–77:6. DOI: 10.1145/2593069.2593090 (cit. on p. 83).

[33] Jason Cong, Chang Wu, and Yuzheng Ding. "Cut Ranking and Pruning: Enabling a General and Efficient FPGA Mapping Solution". In: *Proceedings of the 7th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, USA, Feb. 1999, pp. 29–35. DOI: 10.1145/296399.296425 (cit. on p. 76).

[34] Nitin Dahad. *Cadence Speeds Billion Gate SoC Verification*. https://www.embedded.com/cadence-speeds-billion-gate-soc-verification/. Retrieved May 2023. Apr. 2021 (cit. on p. 137).

[35] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawaj, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald G. Dreslinski, Christopher Batten, and Michael B. Taylor. "The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips". In: *IEEE Micro* 38.2 (Apr. 2018), pp. 30–41. DOI: 10.1109/MM.2018.022071133 (cit. on p. 3).

[36] Monty Denneau. "The Yorktown Simulation Engine". In: *Proceedings of the 19th Design Automation Conference*. Las Vegas, NV, USA, June 1982, pp. 55–59. DOI: 10.1145/800263.809186 (cit. on p. 138).

[37] Charles M. Fiduccia and Robert M. Mattheyses. "A Linear-Time Heuristic for Improving Network Partitions". In: *Proceedings of the 19th Design Automation Conference*. Las Vegas, NV, USA, June 1982, pp. 175–181. DOI: 10.1145/800263.809204 (cit. on p. 98).

[38] Harry Foster. *Part 4: The 2020 Wilson Research Group Functional Verification Study, FPGA Verification Effort Trends*. https://blogs.sw.siemens.com/verificationhorizons/2020/12/02/part-4-the-2020-wilson-research-group-functional-verification-study/. Retrieved Jan 2023. Dec. 2020 (cit. on p. 11).

[39] Harry Foster. *Part 8: The 2020 Wilson Research Group Functional Verification Study, IC/ASIC Resource Trends*. https://blogs.sw.siemens.com/verificationhorizons/2020/12/02/part-4-the-2020-wilson-research-group-functional-verification-study/. Retrieved Jan 2023. Siemens, Jan. 2021 (cit. on p. 11).

[40] Werner Geurts, Francky Catthoor, Serge Vernalde, and Hugo ManChen. *Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications*. New York, NY, USA: Springer, 1996. ISBN: 9780792398202 (cit. on p. 77).

[41] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. "Dynamically Specialized Datapaths for Energy Efficient Computing". In: *Proceedings of the 17th International Symposium on High-Performance Computer Architecture*. San Antonio, Texas, USA, Feb. 2011, pp. 503–514. DOI: 10.1109/HPCA.2011.5749755 (cit. on p. 65).

[42] Mark Harris. *GPU Pro Tip: CUDA 7 Streams Simplify Concurrency*. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/. Retrieved Feb 2023. NVIDIA, Jan. 2015 (cit. on p. 25).

[43] Mark Harris and Kyrylo Perelygin. *Cooperative Groups: Flexible CUDA Thread Programming*. https://developer.nvidia.com/blog/cooperative-groups/. Retrieved Apr 2023. NVIDIA, Oct. 2017 (cit. on p. 25).

[44] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 6th Edition*. San Francisco, CA, USA: Morgan Kaufmann, 2017. ISBN: 9780128119051 (cit. on p. 2).

[45] *High throughput JPEG decoder*. https://github.com/ultraembedded/core_jpeg. Oct. 2020 (cit. on pp. 83, 86).

[46] Matthew Hofmann, Zhiyao Tang, Jonathan Orgill, Jonathan Nelson, David Glanzman, Brent Nelson, and André DeHon. "XBERT: Xilinx Logical-Level Bitstream Embedded RAM Transfusion". In: *Proceedings of the 29th IEEE International Symposium on Field-Programmable Custom Computing Machines*. Virtual, May 2021, pp. 1–9. DOI: 10.1109/FCCM51124.2021.00009 (cit. on p. 108).

[47] Jim Hogan. *Hogan Compares Palladium, Veloce, EVE ZeBu, Aldec, Bluespec, Dini*. http://www.deepchip.com/items/0522-04.html. Retrieved Jan 2023. Apr. 2018 (cit. on p. 138).

[48] "IEEE Standard for Standard SystemC Language Reference Manual". In: *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* (2012), pp. 1–638. DOI: 10.1109/IEEESTD.2012.6134619 (cit. on p. 140).

[49] Natalie D. Enright Jerger, Tushar Krishna, and Li-Shiuan Peh. *On-Chip Networks, 2nd Edition*. Morgan & Claypool, 2017. ISBN: 9783031006272 (cit. on p. 42).

[50] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. "Dissecting the Graphcore IPU Architecture via Microbenchmarking". In: *arXiv* (Dec. 2019). DOI: 10.48550/arXiv.1912.03413 (cit. on p. 143).

[51] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. "TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings". In: *Proceedings of the 50th International Symposium on Computer Architecture*. Orlando, FL, USA, June 2023, pp. 1181–1194. DOI: 10.1145/3579371.3589350 (cit. on p. 3).

[52] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia

Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *Proceedings of the 44th International Symposium on Computer Architecture*. Toronto, ON, Canada, June 2017, pp. 1–12. DOI: 10.1145/3079856.3080246 (cit. on p. 3).

[53] Nachiket Kapre and Jan Gray. “Hoplite: A Deflection-Routed Directional Torus NoC for FPGAs”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 10.2 (Mar. 2017), 14:1–14:24. DOI: 10.1145/3027486 (cit. on p. 42).

[54] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy H. Katz, Jonathan Bachrach, and Krste Asanovic. “FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud”. In: *Proceedings of the 45th International Symposium on Computer Architecture*. Los Angeles, CA, USA, June 2018, pp. 29–42. DOI: 10.1109/ISCA.2018.00014 (cit. on p. 137).

[55] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. “Multilevel Hypergraph Partitioning: Applications in VLSI Domain”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7.1 (Mar. 1999), pp. 69–79. DOI: 10.1109/92.748202 (cit. on p. 67).

[56] Sahand Kashani, Mahyar Emami, and James R. Larus. “Bitfiltrator: A General Approach for Reverse-Engineering Xilinx Bitstream Formats”. In: *Proceedings of the 32nd International Conference on Field Programmable Logic and Applications*. Belfast, UK, Aug. 2022, pp. 1–8. DOI: 10.1109/FPL57034.2022.00039 (cit. on p. 107).

[57] Sahand Kashani, Mahyar Emami, and James R. Larus. *Bitfiltrator: A General Approach for Reverse-Engineering Xilinx Bitstream Formats*. https://github.com/epfl-vlsc/bitfiltrator. Aug. 2022 (cit. on pp. 108, 129).

[58] Donggyu Kim. *riscv-mini*. https://github.com/ucb-bar/riscv-mini. July 2022 (cit. on p. 83).

[59] Balakrishnan Krishnamurthy. “An Improved Min-Cut Algorithm for Partitioning VLSI Networks”. In: *IEEE Transactions on Computers* 33.5 (May 1984), pp. 438–446. DOI: 10.1109/TC.1984.1676460 (cit. on p. 98).

[60] Tuan Minh La, Kaspar Matas, Nikola Grunchevski, Khoa Dang Pham, and Dirk Koch. “FPGADefender: Malicious Self-Oscillator Scanning for Xilinx UltraScale+ FPGAs”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13.3 (Sept. 2020). DOI: 10.1145/3402937 (cit. on p. 108).

[61] Charles Eric LaForest, Zimo Li, Tristan O’Rourke, Ming G. Liu, and J. Gregory Steffan. “Composing Multi-Ported Memories on FPGAs”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7.3 (Sept. 2014), 16:1–16:23. DOI: 10.1145/2629629 (cit. on pp. 55, 57).

[62] Joo Hwan Lee, Hui Zhang, Veronica Lagrange Moutinho dos Reis, Praveen Krishnamoorthy, Xiaodong Zhao, and Yang-Seok Ki. “SmartSSD: FPGA Accelerated Near-Storage Data Analytics on SSD”. In: *IEEE Computer Architecture Letters* 19.2 (July 2020), pp. 114–117. DOI: 10.1109/LCA.2020.3009347 (cit. on p. 3).

[63] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. "From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus". In: *Proceedings of the 51st International Conference on Parallel Processing*. Bordeaux, France, Aug. 2022. DOI: 10.1145/3545008.3545091 (cit. on pp. 25, 140).

[64] Guillem Lopez-Paradis, Brian Li, Adrila Armejach, Stefan Wallentowitz, Miquel Moreto, and Jonathan Balkind. "Fast Behavioral RTL Simulation of 10B Transistor SoC Designs with Metro-MPI". In: *Proceedings of the 26th Conference on Design, Automation and Test in Europe*. Antwerp, Belgium, Apr. 2023, pp. 1–6. DOI: 10.23919/DATE56975.2023.10137080 (cit. on p. 139).

[65] Kristiyan Manev, Joseph Powell, Kaspar Matas, and Dirk Koch. "byteman: A Bitstream Manipulation Framework". In: *Proceedings of the 21st International Conference on Field Programmable Technology*. Hong Kong, China, Dec. 2022, pp. 1–9. DOI: 10.1109/ICFPT56656.2022.9974549 (cit. on pp. 107, 108).

[66] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Q. Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. "A Hardware-Software Blueprint for Flexible Deep Learning Specialization". In: *IEEE Micro* 39.5 (July 2019), pp. 8–16. DOI: 10.1109/MM.2019.2928962 (cit. on p. 83).

[67] Mahesh Nanjundappa, Hiren D. Patel, Bijoy Antony Jose, and Sandeep K. Shukla. "SCGPSim: A Fast SystemC Simulator on GPUs". In: *Proceedings of the 15th Asia and South Pacific Design Automation Conference*. Taipei, Taiwan, Jan. 2010, pp. 149–154. DOI: 10.1109/ASPDAC.2010.5419903 (cit. on p. 140).

[68] Rishiyur S. Nikhil. "Bluespec System Verilog: Efficient, Correct RTL From High Level Specifications". In: *Proceedings of the 2nd International Conference on Formal Methods and Models for System Design*. San Diego, CA, USA, June 2004, pp. 69–70. DOI: 10.1109/MEMCOD.2004.1459818 (cit. on p. 138).

[69] Alan Nishioka and Philip Freidin. *Tell me about the .BIT file format*. http://www.fpga-faq.com/FAQ_Pages/0026_Tell_me_about_bit_files.htm. Retrieved Jan 2022. Nov. 2001 (cit. on p. 111).

[70] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. "A General Constraint-Centric Scheduling Framework for Spatial Architectures". In: *Proceedings of the 34th International Conference on Programming Language Design and Implementation*. Seattle, WA, USA, June 2013, pp. 495–506. DOI: 10.1145/2491956.2462163 (cit. on p. 65).

[71] *NVIDIA Ada GPU Architecture*. Version 2.01. NVIDIA. Apr. 5, 2023 (cit. on p. 3).

[72] *NVIDIA Mellanox ConnectX-6 SmartNIC Adapter*. https://www.nvidia.com/en-us/networking/ethernet/connectx-6/. Retrieved May 2023. NVIDIA (cit. on p. 3).

[73] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. *An Overview of the Scala Programming Language*. Tech. rep. Lausanne, Switzerland: EPFL, 2006 (cit. on p. 48).

[74] *Open-Source FPGA Bitcoin Miner*. https://github.com/progranism/Open-Source-FPGA-Bitcoin-Miner. July 2013 (cit. on p. 83).

[75] Khoa Dang Pham, Edson Horta, and Dirk Koch. "BitMan: A Tool and API for FPGA Bitstream Manipulations". In: *Proceedings of the 20th Conference on Design, Automation and Test in Europe*. Lausanne, Switzerland, Mar. 2017, pp. 894–897. DOI: 10.23919/DATE.2017.7927114 (cit. on pp. 108, 109, 111, 126).

[76] Clément Pit-Claudel, Thomas Bourgeat, Stella Lau, Arvind, and Adam Chlipala. "Effective Simulation and Debugging for a High-Level Hardware Language Using Software Compilers". In: *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems*. Virtual, Apr. 2021, pp. 789–803. DOI: 10.1145/3445814.3446720 (cit. on p. 138).

[77] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services". In: *Communications of the ACM* 59.11 (Oct. 2016), pp. 114–122. DOI: 10.1145/2996868 (cit. on p. 3).

[78] Hao Qian and Yangdong Deng. "Accelerating RTL Simulation with GPUs". In: *Proceedings of the 30th International Conference on Computer-Aided Design*. San Jose, CA, USA, Nov. 2011, pp. 687–693. DOI: 10.1109/ICCAD.2011.6105404 (cit. on p. 140).

[79] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. "AI and ML Accelerator Survey and Trends". In: *Proceedings of the 2022 High Performance Extreme Computing Conference*. Waltham, MA, USA, Sept. 2022, pp. 1–10. DOI: 10.1109/HPEC55821.2022.9926331 (cit. on p. 4).

[80] Lauro Rizzatti. *Budgeting for Hardware Emulation Platforms*. https://www.gsaglobal.org/forums/budgeting-for-hardware-emulation-platforms/. Retrieved Jan 2023. 2020 (cit. on p. 138).

[81] Lauro Rizzatti and Charley Selvidge. *Designing a Modern Hardware Emulation Platform*. https://www.electronicdesign.com/technologies/test-measurement/article/21120302/designing-a-modern-hardware-emulation-platform. Retrieved Jan 2023. Jan. 2020 (cit. on pp. 137, 138).

[82] Andrew Rushton. *VHDL for Logic Synthesis, 3rd Edition*. Chichester, UK: John Wiley & Sons, 2011. ISBN: 9780470688472 (cit. on p. 14).

[83] Laura A. Sanchis. "Multiple-Way Network Partitioning". In: *IEEE Transactions on Computers* 38.1 (Jan. 1989), pp. 62–81. DOI: 10.1109/12.8730 (cit. on p. 98).

[84] Vivek Sarkar and John L. Hennessy. "Compile-Time Partitioning and Scheduling of Parallel Programs". In: *Proceedings of the 1986 International Conference on Compiler Construction*. Palo Alto, CA, USA, June 1986, pp. 17–26. DOI: 10.1145/12276.13313 (cit. on p. 81).

[85] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. “High-Quality Hypergraph Partitioning”. In: *ACM Journal of Experimental Algorithmics* 27 (Feb. 2023), 1:9–1:39. DOI: 10.1145/3529090 (cit. on p. 67).

[86] Ofer Shacham and Masumi Reynders. *Pixel Visual Core: Image Processing and Machine Learning on Pixel 2*. https://blog.google/products/pixel/pixel-visual-core-image-processing-and-machine-learning-pixel-2/. Retrieved May 2023. Google, Oct. 2017 (cit. on p. 3).

[87] Ken Shirriff. *Reverse-Engineering the First FPGA Chip, the XC2064*. http://www.righto.com/2020/09/reverse-engineering-first-fpga-chip.html. Retrieved Mar 2022. Sept. 2020 (cit. on p. 108).

[88] *Simulation speed for rocket-chip: VCS vs Verilator #2160*. https://github.com/chipsalliance/rocket-chip/issues/2160. Oct. 2019 (cit. on p. 81).

[89] Steven P. Smith, M. Ray Mercer, and B. Brodk. “Demand Driven Simulation: BACKSIM”. In: *Proceedings of the 24th Design Automation Conference*. Miami Beach, FL, USA, July 1987, pp. 181–187. DOI: 10.1145/37888.37915 (cit. on p. 14).

[90] Wilson Snyder. “Verilator 4.0: Open Simulation Goes Multithreaded”. 2018 Open Source Digital Design Conference. Gdansk, Poland, Sept. 22, 2018. URL: https://veripool.org/papers/Verilator_v4_Multithreaded_OrConf2018.pdf (cit. on p. 81).

[91] Wilson Snyder. “Verilator, Accelerated: Accelerating Development, and Case Study of Accelerating Performance”. 2nd Workshop on Open-Source Design Automation (OSDA). Grenoble, France, Mar. 13, 2020. URL: https://veripool.org/papers/Verilator_Accelerated_OSDA2020.pdf (cit. on p. 81).

[92] Wilson Snyder. *Welcome to Verilator*. https://veripool.org/verilator/. Retrieved Jan 2023 (cit. on pp. 5, 81).

[93] Tobias Strauch. “Combining Simulation and FPGA Based Verification to an Affordable and Ultra-Fast Multi-Billion-Gate Verification System”. In: *Proceedings of the 30th International Workshop on Rapid System Prototyping*. New York, NY, USA, Oct. 2019, pp. 22–28. DOI: 10.1145/3339985.3358487 (cit. on p. 139).

[94] Keichi Takahashi, Soya Fujimoto, Satoru Nagase, Yoko Isobe, Yoichi Shimomura, Ryusuke Egawa, and Hiroyuki Takizawa. “Performance Evaluation of a Next-Generation SX-Aurora TSUBASA Vector Supercomputer”. In: *ARXIV* (Apr. 2023). DOI: 10.48550/arXiv.2304.11921 (cit. on p. 3).

[95] Stephanie Tapp. *XAPP1230: Configuration Readback Capture in UltraScale FPGAs*. Version 1.1. Xilinx. Nov. 20, 2015 (cit. on pp. 117, 128).

[96] Xiang Tian and Khaled Benkrid. “Design and Implementation of a High Performance Financial Monte-Carlo Simulation Engine on an FPGA Supercomputer”. In: *Proceedings of the 7th International Conference on Field Programmable Technology*. Taipei, Taiwan, Dec. 2008, pp. 81–88. DOI: 10.1109/FPT.2008.4762369 (cit. on p. 83).

[97] Yatish Turakhia, Gill Bejerano, and William J. Dally. "Darwin: A Genomics Co-processor Provides up to 15,000X Acceleration on Long Read Assembly". In: *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. Williamsburg, VA, USA, Mar. 2018, pp. 199–213. DOI: 10.1145/3173162.3173193 (cit. on p. 3).

[98] *UG470: 7 Series FPGAs Configuration User Guide*. Version 1.13.1. Xilinx. Aug. 20, 2018 (cit. on p. 125).

[99] *UG570: UltraScale Architecture Configuration User Guide*. Version 1.16. Xilinx. Jan. 14, 2022 (cit. on pp. 108, 111, 112, 126).

[100] *UG572: UltraScale Architecture Clocking Resources*. Version 1.10. Xilinx. Aug. 28, 2020 (cit. on p. 35).

[101] *UG574: UltraScale Architecture Configurable Logic Block User Guide*. Version 1.5. Xilinx. Feb. 28, 2017 (cit. on pp. 35, 110, 127, 128).

[102] *UG908: Vivado Design Suite User Guide, Programming and Debugging*. Version 2021.1. Xilinx. June 16, 2021 (cit. on pp. 115–117).

[103] *UG909: Vivado Design Suite User Guide, Dynamic Function eXchange*. Version 2021.1. Xilinx. June 30, 2021 (cit. on pp. 111, 114, 118).

[104] A. Ullah, E. Sanchez, L. Sterpone, L.A. Cardona, and C. Ferrer. "An FPGA-Based Dynamically Reconfigurable Platform for Emulation of Permanent Faults in ASICs". In: *Microelectronics Reliability* 75 (Aug. 2017), pp. 110–120. DOI: https://doi.org/10.1016/j.microrel.2017.06.032 (cit. on p. 108).

[105] *UltraScale Architecture DSP Slice*. Version 1.10. Xilinx. Sept. 22, 2020 (cit. on p. 35).

[106] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Communications of the ACM* 33.8 (Aug. 1990), pp. 103–111. DOI: 10.1145/79173.79181 (cit. on p. 19).

[107] *Virtual Machine Series*. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series/. Retrieved Oct 2022. Microsoft (cit. on p. 83).

[108] Elliot Waingold, Michael B. Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew I. Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman P. Amarasinghe, and Anant Agarwal. "Baring It All to Software: Raw Machines". In: *IEEE Computer* 30.9 (Sept. 1997), pp. 86–93. DOI: 10.1109/2.612254 (cit. on pp. 5, 40, 140).

[109] David W. Wall. "Limits of Instruction-Level Parallelism". In: *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. Santa Clara, CA, USA, Apr. 1991, pp. 176–188. DOI: 10.1145/106972.106991 (cit. on p. 1).

[110] Haoyuan Wang and Scott Beamer. "RepCut: Superlinear Parallel RTL Simulation with Replication-Aided Partitioning". In: *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems*. Vancouver, BC, Canada, Mar. 2023, pp. 572–585. DOI: 10.1145/3582016.3582034 (cit. on p. 139).

[111] Zhicheng Wang and Peter M. Maurer. "LECSIM: A Levelized Event Driven Compiled Logic Simulation". In: *Proceedings of the 27th Design Automation Conference*. Orlando, FL, USA, June 1990, pp. 491–496. DOI: 10.1145/123186.123349 (cit. on p. 15).

[112] Christian Wimmer and Michael Franz. “Linear Scan Register Allocation on SSA Form”. In: *Proceedings of the 8th International Symposium on Code Generation and Optimization*. Toronto, ON, CA, Apr. 2010, pp. 170–179. doi: 10.1145/1772954.1772979 (cit. on p. 79).

[113] Claire Wolf. *Yosys Open SYnthesis Suite*. https://yosyshq.net/yosys/ (cit. on p. 61).

[114] William G. Wong. *Lauro Rizzatti Explains Hardware Emulation’s Appeal*. https://www.electronicdesign.com/technologies/eda/article/21800052/lauro-rizzatti-explains-hardware-emulations-appeal. Retrieved Jan 2023. July 2014 (cit. on p. 137).

[115] Tao Yang and Apostolos Gerasoulis. “DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors”. In: *IEEE Transactions on Parallel and Distributed Systems* 5.9 (Sept. 1994), pp. 951–967. doi: 10.1109/71.308533 (cit. on p. 98).

[116] Yanqing Zhang, Haoxing Ren, and Brucek Khailany. “Opportunities for RTL and Gate Level Simulation using GPUs”. In: *Proceedings of the 39th International Conference on Computer-Aided Design*. San Diego, CA, USA, Nov. 2020, 166:1–166:5. doi: 10.1145/3400302.3415773 (cit. on p. 140).

Sahand Kashani

Goal

Identify bottlenecks in modern computing systems, then design novel hardware and compiler stacks to elegantly circumvent them. I wish to contribute to high-performance systems that solve challenging real-world problems.

Research Interests

Compilers, computer architecture, reconfigurable systems, accelerators

Education

2017–2023 **PhD in Computer Science**, *EPFL*, Lausanne/Switzerland
Thesis: Building Chips Faster: Hardware-Compiler Co-Design for Accelerated RTL Simulation
Advisor: Prof. James R. Larus

2014–2017 **Master in Computer Science**, *EPFL*, Lausanne/Switzerland
Specialization: Computer Engineering

2010–2014 **Bachelor in Computer Science**, *EPFL*, Lausanne/Switzerland

Professional Experience

2019 **Research Internship**, *Microsoft Research*, Redmond/USA
Designed and implemented a new control processor for Microsoft's BrainWave neural network inference service. The new design improves throughput in critical loops of large BrainWave programs by $3\times$.

2017 **Master Thesis**, *Syderal Space & EPFL*
Developed an FPGA overlay that leverages partial reconfiguration to deploy hardware accelerators in satellites.

2016 **Research Internship**, *Processor Architecture Laboratory (EPFL)*, Lausanne/Switzerland
Analyzed recurring patterns in logic functions to inspire the design of more efficient FPGA logic blocks.

2015 **Engineering Internship**, *senseFly SA*, Cheseaux-sur-Lausanne/Switzerland
Designed an FPGA interface and Linux driver for a drone-mounted thermal camera.

2014 **Engineering Internship**, *Processor Architecture Laboratory (EPFL)*, Lausanne/Switzerland
Evaluated the performance of processor-to-FPGA communication in Altera Cyclone V SoC-FPGA devices.

2013 **Engineering Internship**, *Barclays Wealth*, Geneva/Switzerland
Automated the quality control process for code written by junior developers.

Publications

ASPLOS'24 Manticore: Hardware-Accelerated RTL Simulation with Static Bulk-Synchronous Parallelism
Mahyar Emami*, **Sahand Kashani***, Keisuke Kamahori, Sepehr Pourghannad, James R. Larus
29th International Conference on Architectural Support for Programming Languages and Operating Systems
San Diego, USA, April 2024
***equal contribution**

FPL'22 Bitfiltrator: A General Approach for Reverse-Engineering Xilinx Bitstream Formats
Sahand Kashani, Mahyar Emami, James R. Larus
32nd International Conference on Field-Programmable Logic and Applications
Belfast, UK, August 2022
Best Paper Award

LATTE'21 Compile-Time RTL Interpreters
Sahand Kashani, James R. Larus
First Workshop on Languages, Tools, and Techniques for Accelerator Design
Virtual, April 2021

AACBB'19 IMPACT: Interval-based Multi-pass Proteomic Alignment with Constant Traceback
Sahand Kashani, Stuart Byma, James R. Larus
2nd HPCA Workshop on Accelerator Architectures in Computational Biology and Bioinformatics
Washington DC, USA, February 2019

Snap-On User-Space Manager for Dynamically Reconfigurable System-on-Chips
Andrea Guerrieri, **Sahand Kashani**, Mikhail Asiatici, Paolo Ienne
IEEE Access, vol. 7, 2019

AHS'18 A Dynamically Reconfigurable Platform for High-Performance and Low-Power On-Board Processing
Andrea Guerrieri, **Sahand Kashani**, Pasquale Lombardi, Bilel Belhadj, Paolo Ienne
2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)
Edinburgh, UK, August 2018

Books

Fundamentals of System-on-Chip Design on Arm Cortex-M Microcontrollers
René Beuchat, Florian Depraz, Andrea Guerrieri, **Sahand Kashani**
Arm Education Media, 2021

Awards

2022 **Best Paper Award** at the Intl. Conference on Field-Programmable Logic and Applications (FPL)

2021 **Teaching Assistant Award** for embedded systems courses

2016 **Teaching Assistant Award** for software engineering course

2014 **Student Assistant Award** for computer architecture courses

2013 **Introduction to Computer Graphics Project Award** for a cloth simulation project

Skills

Programming

Good C/C++, Java, Python, Scala, VHDL, Verilog
Familiar Assembly, CUDA, JavaScript, Perl, Spark, SQL, Tcl

Compilers

FIRRTL, Yosys, Verilog-to-Routing

EDA Tools

Intel Quartus, Xilinx Vivado, ModelSim, Verilator, Synopsys Design Compiler

General

Good writing and presenting skills.
Critical thinker who can work both autonomously and in a collaborative environment.

Languages

Native English, French
Intermediate German (B2 level)