Technical Report

UCAM-CL-TR-979
ISSN 1476-2986

Number 979

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Efficient virtual cache coherency for multicore systems and accelerators

Xuan Guo

February 2023

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

© 2023 Xuan Guo

This technical report is based on a dissertation submitted September 2022 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Peterhouse College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Efficient Virtual Cache Coherency for Multi-core Systems and Accelerators

Xuan Guo

There is a paradigm shift from general-purpose cores to specialised hardware, which has vastly different programming models. It will be helpful if the existing programming model can be kept and new hardware can co-exist and cooperate with existing userspace software. A virtual cache coherence protocol can be helpful for such task, allowing individual components to perform virtual address accesses without having to include their own hardware for address translation and memory protection. This thesis presents such a protocol, together with tooling and hardware infrastructure that are developed in the process of creating it.

This thesis makes three contributions. The first contribution is in the area of processor simulation techniques. A high-performance simulator is presented for exploring just the behaviour of translation lookaside buffers (TLBs). This is then extended to provide fast cycle-level simulator. The simulator employs an innovative technique to combine binary translation with cycle-level simulation and therefore significantly speedup the simulation process compared to traditional interpretation-based simulators. The simulator built, R2VM, can achieve $\sim$30 million instructions per second (MIPS) in cycle-level simulation in lockstep execution mode, more than 100x the performance of gem5 in a similar mode of operation. For non-cycle-level fast-forward execution, R2VM can achieve $>$400 MIPS per thread. This is significantly faster than gem5's 3 MIPS fast-forward execution, and even better than emulators that exploit dynamic binary translation (DBT), such as QEMU.

The second contribution is a collection of open-source processor and system-on-chip (SoC) components, called Muntjac. Muntjac contains implementation of a RISC-V (RV64GC) core with machine and supervisor privilege levels, as well as cache subsystems and interconnect components that utilise TileLink. An untethered Linux-capable example SoC is implemented with these components. Muntjac core can achieve Dhrystone score of 2.17 DMIPS/MHz and CoreMark score of 3.01 CoreMark/MHz. Muntjac is designed to be

modular, verifiable and extendable, and be a good starting point for education, research and industrial applications.

The final contribution is a virtual cache coherence protocol that permits the use of virtually-indexed virtually-tagged (VIVT) L1 caches. The protocol is designed to allow commonly used read-only synonyms to reside in caches while still maintaining correctness in hardware when writable synonyms occur. The protocol is designed and described in detail, and implemented and evaluated on a field programmable gate array (FPGA) using Muntjac. Caches that communicate using the protocol are implemented, and support has been added to a Linux kernel port. Systems with the protocol have lower resource utilisation and higher maximum frequency compared to the physically coherent counterpart as the TLB is removed from the L1 and the critical path of memory access, while still being comparable in terms of performance per MHz. The flexibility and advantages of the protocol are demonstrated by the creation and integration of easy-to-use accelerators that can be accessed from the general-purpose cores with a low latency.

Acknowledgements

First and foremost, I would like to thank my PhD supervisor Prof. Robert Mullins. Rob gave me tremendous support and guidance throughout the past 7 years, first as the Director of Studies of my undergraduate studies and then as my PhD supervisor. I have always been interested in a very broad range of computer science topics, but my specific interest towards Computer Architecture was sparked by a summer internship at the Computer Laboratory in 2016 organised by Rob. I wrote my first line of SystemVerilog during that internship; the number of SystemVerilog lines written has since been on a magnitude of 100,000. Rob has spent countless hours on my PhD, discussing weekly progress, providing suggestions for drafts and fixing EDA tool license issues. Apart from academic support, Rob has also been a very helpful mentor for my life and career. I will be forever grateful for his support.

I would like to thank my college, Peterhouse, for supporting me financially with the Peterhouse Graduate Studentship. I would also like to thank my college tutors, Dr Saskia Murk-Jansen for being my undergraduate tutor and Dr Christopher Lester for being my graduate tutor, for providing me support on tutorial matters and securing and extending my studentship. I would also like to thank Prof. Timothy Jones, who has served as the second advisor of my PhD and was the supervisor for my Part II thesis. The binary translator work I did in my Part II is very valuable for my PhD.

I would like to thank Dr Daniel Bates. Daniel has been collaborating with me on Muntjac. His expertise in testing and verification has been very helpful to speed up the development process of Muntjac. I would also like to express my gratitude to Alex Bradbury and Gavin Ferris, figures behind lowRISC CIC which has also been a collaborating partner for Muntjac.

I would like to thank Dr Yiren Zhao, my labmate and friend who shares an office with me for 3 years. Discussion in the office has always been insightful and sparks many ideas. I would also like to appreciate the occasional advice given by Dr Junyi Liu when we are having hotpots together. I would also like to thank my best friends, especially Qiujia Li and Yudong Chen, for forming a support bubble with me during the COVID-19 pandemic. The pandemic and the related lockdown have been very challenging, stressful and depressive, and I could not imagine going through this tough period without them. I greatly cherish all my

friendship, and hope that the pandemic could be over soon and I could meet up with other old friends and maybe make new friends.

Finally, I would express my gratitude to my parents and grandparents. Their unconditional love and support have been a constant source of encouragement and motivation for me.

Table of Contents

List of Figures 11

List of Tables 15

List of Listings 17

Acronyms 19

Terminology 23

1 Introduction 25

1.1 Hypothesis 27
1.2 Contributions 28
1.3 Thesis Outline 29
1.4 Publications 30

2 Background 31

2.1 Virtual Memory Support for Accelerators 31
2.2 Cache Coherent Accelerators 32
2.3 Sharing Address Translation Hardware 33
2.3.1 ASID Space 33
2.3.2 Shared TLB 33
2.4 Virtual Cache 34
2.5 Existing Solutions for Synonyms 34
2.5.1 Synonym Prevention 34
2.5.2 Synonym Avoidance 35
2.5.3 Physical Tagging 36
2.5.4 Reverse Mapping and L2 Backpointer 37
2.5.5 Private Physical L2 38

2.5.6 Synonym Remapping 39
2.5.7 Hybrid Caching 40
2.5.8 Intermediate Address Space 41
2.5.9 Write-Through 42
2.6 RISC-V 43
2.6.1 Virtual Memory Addressing in RISC-V 43
2.6.2 ASID Space 44
2.7 Instruction Set Simulators 45
2.7.1 Binary Translation 46
2.7.2 Multi-core Simulation 46
2.8 Benchmarks 47

3 High-performance Simulation Techniques for Memory System Exploration 49
3.1 TLB Simulation 49
3.1.1 Simulator Design 49
3.1.2 Simulator Implementation 51
3.1.3 Simulator Performance 53
3.2 Software Validation and Profiling 54
3.2.1 Software Validation 54
3.2.2 Software Profiling 55
3.3 Shared TLB and Global ASID 56
3.3.1 Shared TLB 56
3.3.2 Global ASID 60
3.3.3 ASID Sharing Extension 60
3.4 Classifying Synonyms 62
3.5 R2VM – Fast Cycle-Level Instruction Simulator 63
3.5.1 Implementation Overview 64
3.5.2 Pipeline Simulation 66
3.5.3 Multi-core Simulation 67
3.5.4 Memory Simulation 70
3.5.5 Runtime Reconfiguration 74
3.5.6 Other Techniques and Design Decisions 75
3.5.7 Evaluation 80

4 A Practical Virtual Cache Coherence Protocol 83
4.1 Design Principles 83
4.2 Imperfect Reverse Translation and Batch Invalidation 85

4.3 Simulation Technique 88
4.4 Evaluation Technique 92
4.4.1 Baseline 92
4.4.2 Benchmarks 93
4.5 Dirty Cache Lines and Page Table Entry Changes 94
4.6 Dirty Cache Lines and ASID Changes 97
4.7 L2 Eviction of Read-only Synonyms 98
4.7.1 ASID Batch Invalidation 99
4.7.2 OS Assisted Synonym Placement 100
4.7.3 Partial Batch Invalidation 101
4.7.4 Bloom Filter 103
4.8 Summary of the Protocol Design after Simulation 108

5 Hardware Baseline and Virtual Cache Protocol Implementation 115
5.1 Hardware Baseline 115
5.1.1 Processor Baseline 115
5.1.2 Protocol Baseline 116
5.2 Muntjac – Open Source Collections of Processor and SoC Components . . 124
5.2.1 Core Overview 125
5.2.2 Pipeline Design 127
5.2.3 Cache Design 131
5.2.4 Multi-core Support 138
5.2.5 Testing 141
5.2.6 Performance Characteristics 142
5.3 Virtual Cache Protocol Implementation 147
5.3.1 Protocol Overview 147
5.3.2 Dirty Cache Lines of Global Pages 148
5.3.3 Exclusive Cache Lines and Page Table Entry Changes 150
5.3.4 Dirty Cache Lines and Page Table Walking 151
5.3.5 Dirty Cache Lines and the Accessed Bit 152
5.3.6 Signal Descriptions 153
5.3.7 L1 Cache Design 157
5.3.8 L2 Cache Design 159
5.4 Evaluation 161
5.4.1 Utilisation 162
5.4.2 Memory Latency Testing 162
5.4.3 PARSEC Benchmark 163

5.4.4 Minibench . . . 166
5.4.5 Conclusion . . . 166

6 Virtual Cache Coherency with Accelerators 169
6.1 Baseline . . . 169
6.1.1 Evaluation Technique . . . 169
6.1.2 Baseline Accelerator . . . 170
6.1.3 FlexAcc . . . 172
6.1.4 Summary of Baselines . . . 173
6.2 Non-coherent Accelerators . . . 173
6.2.1 Virtual Addressing . . . 174
6.2.2 Evaluation . . . 175
6.3 Coherent Accelerators . . . 179
6.3.1 Futex-based Completion Notification . . . 179
6.3.2 Using a Core as an Accelerator . . . 184

7 Conclusion and Future Work 187
7.1 Future Work . . . 188
7.1.1 Virtualisation-based Memory Access Acceleration in Full-system Binary Translation . . . 188
7.1.2 Just-in-time Compilation of Advanced Pipeline Models . . . 188
7.1.3 Extensions to Muntjac . . . 189
7.1.4 Multi-level Virtual Cache . . . 189
7.1.5 Distributed Shared Caches with the Virtual Cache Protocol . . . 189
7.1.6 Multi-user and Time-shared Accelerator . . . 191
7.1.7 Many-core Systems with Virtual Cache Coherency . . . 191
7.1.8 Seas of Accelerators with Virtual Cache Coherency . . . 191

References 193

List of Figures

2.1 Flow in virtual cache system using reverse maps 37

3.1 Control flow of a guest data memory access in TLBSim 52
3.2 Performance of QEMU, with and without TLB simulation 54
3.3 Local L2 TLB miss rates running the PARSEC benchmarks 58
3.4 Local L2 TLB miss rates compiling the Linux kernel 59
3.5 Local L2 TLB miss rates in relation to per-core L2 TLB size 59
3.6 High-level overview of R2VM components 65
3.7 Control flow overview of R2VM 65
3.8 Memory layout of fibers 68
3.9 Control flow for memory accesses 70
3.10 Memory layout of a tag entry in the L0 data cache 71
3.11 Performance comparison between models and other simulators 82

4.1 L1 miss flow when the cache line is neither owned nor shared 86
4.2 L1 miss flow when the cache line is owned/shared, and synonym is not present 87
4.3 L1 miss flow when the cache line is shared with synonyms present 87
4.4 A typical flow of the cache coherence protocol being simulated 90
4.5 A typical flow in the simulation implementation of the cache coherence protocol 91
4.6 PARSEC benchmark performance of evaluated PTE change options 95
4.7 Number of dirty/exclusive cache lines when the page is evicted from TLB . 96
4.8 PARSEC benchmark performance of evaluated ASID change options 98
4.9 Minibench evaluation of ASID batch invalidation 100
4.10 Addresses and masks for different types of synonyms in partial batch invalidation 102
4.11 Encoding of different types of synonyms in partial batch invalidation 102
4.12 Minibench evaluation of partial batch invalidation 103

4.13 Comparison of results from simulation, random process sampling and analytical solution of the mean number of accesses to an invalidation for Bloom filter usage . . . 106
4.14 Mean number of accesses to an invalidation . . . 106
4.15 Minibench evaluation of the Bloom filter optimisation . . . 107
4.16 PARSEC benchmark performance of evaluated L2 synonym eviction options 108
4.17 Metadata layout of an L1 cache line . . . 108
4.18 Metadata layout of an L2 cache line . . . 109
4.19 PARSEC benchmark performance comparison between physical, virtual cache and their variants . . . 111
4.20 Summary of minibench evaluation with cache line granularity . . . 112
4.21 Summary of minibench evaluation with page granularity . . . 112

5.1 Ambiguous TileLink transaction if E channel is absent . . . 121
5.2 Example multi-core Muntjac system . . . 124
5.3 High-level overview of Muntjac core components . . . 126
5.4 Muntjac frontend design . . . 127
5.5 Muntjac backend design . . . 129
5.6 Muntjac FPU design . . . 130
5.7 Muntjac L1 data cache subcomponents . . . 132
5.8 Muntjac L1 data cache fast path . . . 132
5.9 Alternative Muntjac L1 data cache fast path design . . . 133
5.10 Muntjac L1 instruction cache subcomponents . . . 135
5.11 Muntjac L1 instruction cache fast path . . . 136
5.12 Parallel data SRAM lookup across multiple ways, given a fixed offset . . . 137
5.13 Wide data SRAM access of multiple words within the same cache line . . . 137
5.14 Simple broadcasting bus to bridge TL-C multiple hosts to a TL-UH port . . 139
5.15 Muntjac L2 cache design . . . 140
5.16 Memory access latency vs working set size . . . 144
5.17 Bus transactions when semaphore spinning is performed in the S state . . . 145
5.18 Bus transactions when semaphore spinning is performed in the M state . . . 146
5.19 High-level component overview of the implementation of the virtual cache protocol . . . 147
5.20 Representation of global pages in root page tables under normal circumstances in Linux . . . 149
5.21 Representation of global pages after root PTE is changed in Linux . . . 149
5.22 Protocol transactions when a page table walk happens for a writeback message 152

5.23 Fast path of the virtual L1 data cache design 157
5.24 Components of the virtual L1 data cache design 158
5.25 Fast path of the virtual L1 instruction cache design 158
5.26 Components of the virtual L1 instruction cache design 158
5.27 Memory management unit of the virtual L2 cache 160
5.28 L2 cache bank design under the proposed virtual protocol 161
5.29 Memory access latency vs working set size 163
5.30 PARSEC benchmark performance comparison between physical, virtual cache implementations and their variants 164
5.31 PARSEC benchmark performance comparison between physical, virtual cache implementations and their variants, with 1-cycle delay added to the physical caches 165
5.32 PARSEC benchmark performance comparison of virtual cache implementations, with inter-core TLB entry sharing disabled 165
5.33 Summary of minibench evaluation on hardware implementation with cache line granularity 166
5.34 Summary of minibench evaluation on hardware implementation with page granularity 167

6.1 Accelerator implementation 171
6.2 FlexAcc design 172
6.3 Virtually addressed accelerator framework 175
6.4 Accelerator throughput in relation to chunk size 176
6.5 Time spent per cache line, in relation to chunk size 177
6.6 Accelerator throughput for streaming workload 178
6.7 Time spent for each request for streaming workload 178
6.8 Virtually coherent accelerator framework 181
6.9 Accelerator throughput of futex-based completion notification 182
6.10 Accelerator throughput of futex-based completion notification for streaming workload 183
6.11 Accelerator throughput of core with integrated accelerator 186

7.1 Possible system design with distributed LLC using virtual cache protocol . 190

List of Tables

3.1 SFENCE.VMA instructions issued by the Linux kernel 55
3.2 Look-up table for whether translation caches could be shared 61
3.3 List of pre-implemented pipeline models 80
3.4 List of pre-implemented memory system models 80

4.1 Comparison between physical (PIPT) caches and different virtual cache techniques 89

5.1 Summary of interconnect protocol differences 123
5.2 List of RISC-V instruction-set extension standards implemented 126
5.3 Muntjac synthesis result for Xilinx Kintex 7 with Synopsys Synplify 142
5.4 Number of stall-cycles for different instructions 143
5.5 Signal lists of the A channel 153
5.6 Signal lists of the D channel 154
5.7 Signal lists of the E channel 154
5.8 Signal lists of the C channel 155
5.9 Signal lists of the B channel 155
5.10 Muntjac synthesis result for Xilinx Kintex 7 with Synopsys Synplify, with virtual cache 162

6.1 States and actions of an example futex-based mutex implementation 180

List of Listings

3.1 Abstract class shared by TLB components in TLBSim 51
3.2 Timing simple model implementation 66
3.3 Example of generated code with yield calls 68
3.4 Implementation of the fiber yielding code 69
3.5 Generated assembly sequence for memory load 72
3.6 Generated assembly sequence for memory store 73
3.7 Example assembly changes with block chaining, before and after patching . 77
3.8 Example patched assembly with speculative block chaining 78
3.9 Example of generated assembly for a cross-page instruction 79
5.1 Assembly sequence for the `DOWN` operation of a semaphore that spins in the S state 144
5.2 Assembly sequence for the `DOWN` operation of a semaphore that spins in the M state 146
6.1 API and example usage of the accelerator core 185

Acronyms

ALU arithmetic logic unit. 129, 133

AMOALU atomic memory operation ALU. 133

API application programming interface. 17, 26, 184, 185, 188

ASID address space identifier. 11, 29, 33, 34, 44, 52, 54, 55, 57, 58, 60–62, 85, 88, 93, 97–101, 103, 109, 110, 148, 150, 153–157, 174, 187, 189

ASLR address space layout randomization. 62, 93, 101

BTB branch target buffer. 128

CAS compare-and-swap. 91

CLINT core local interrupt. 138, 161

CPU central processing unit. 25–28, 32, 34, 42, 45, 53, 56, 67, 117, 140, 141, 173, 181, 183, 188

CSR control and status register. 44, 60–62, 74, 81, 97, 129, 143, 184, 189

DBT dynamic binary translation. 3, 29, 46, 49, 50, 53, 67, 69–71, 73–75, 82, 187, 188

DMA direct memory access. 26, 27, 31, 32, 56, 122, 138, 170, 171, 174, 175, 177, 184, 186

DSL domain-specific language. 116, 188

EDA electronic design automation. 116

FPGA field programmable gate array. 4, 26, 28, 92, 105, 115, 143, 161, 166

FPU floating point unit. 12, 81, 130, 131, 142, 161, 162, 184

futex fast userspace mutex. 179–183

GPGPU general-purpose GPU. 25

GPU graphics processing unit. 25, 26, 42, 56, 60, 171, 173

HDL hardware description language. 115

IOMMU I/O memory management unit. 31, 32

IPI inter-processor interrupt. 46

ISA instruction set architecture. 25, 33, 35, 43, 45, 46, 49–51, 55, 84, 94, 115, 117

JIT just-in-time. 188

LLC last-level cache. 13, 32, 84, 119, 173, 189, 190

LPAR logical partitioning. 60–62

LR/SC load-reserved/store-conditional. 43, 84, 118, 123, 134, 147

LRU least-recently used. 50, 73, 74

LUT lookup table. 142, 162, 183

MESI modified, exclusive, shared and invalid. 80–82, 85, 92, 94, 119, 123

MIPS million instructions per second. 3, 28, 53, 54, 64, 81, 187

MMIO memory-mapped I/O. 170, 173, 184

MMU memory management unit. 28, 39, 41, 86, 147, 153, 154, 156, 157, 159, 160, 162–164, 166, 175, 176, 189, 190

MSHR miss status holding register. 115

MSI modified, shared and invalid. 42, 84

mutex mutual exclusion. 90, 91

OoO out-of-order. 36, 116

OS operating system. 29, 31, 32, 35, 36, 39–43, 52, 54, 57, 74, 85, 94, 96, 97, 100, 101, 103, 115, 125, 131, 150, 152, 153, 157, 172, 174, 181, 183, 184, 187

PC program counter. 46, 64, 76

PIPT physically-indexed physically-tagged. 15, 34, 36, 38, 89, 133

PLIC platform-level interrupt controller. 138, 161

PTE page table entry. 11, 12, 35, 39, 54, 55, 57, 88, 90, 95–97, 109, 138, 148–152, 174

PTW page table walker. 27, 56, 57, 86, 97, 151, 157, 159

RAS return address stack. 128

RTL register-transfer level. 45, 53, 80, 81, 115

SBI supervisor binary interface. 66, 161

SMP simultaneous multiprocessing. 43

SoC system-on-chip. 3, 26, 29, 43, 161–163, 171

syscall system call. 66, 179–182

TLB translation lookaside buffer. 3, 4, 11, 13, 17, 27–29, 31–34, 36, 38–42, 44, 49–63, 70, 71, 73–75, 80, 81, 83–86, 90, 92, 94–97, 108, 110, 111, 131, 133, 134, 138, 142, 144, 151–153, 157, 159–166, 172–176, 187–190

UVM unified virtual-memory. 31

VIPT virtually-indexed physically-tagged. 34, 36, 103

VIVT virtually-indexed virtually-tagged. 4, 28, 32, 34, 36, 184, 188

VMA virtual memory area. 41, 42

Terminology

Historically, for asymmetric communication or control protocols, master/slave terminology is widely used. Cache coherence protocols, being asymmetrical, therefore often employ such terminology in both industrial application and academic literature. The term “master” refers to agents that actively initiate transactions, such as processors, and the term “slave” refers to agents that passively respond to these transactions, such as memory controllers. The usage is widespread and can be seen in PCIe specification [94], AXI specification [5] and TileLink specification [108]. In this thesis, I will avoid these terms and refer the agents as “host” and “device” instead of “master” and “slave”.

Chapter 1

Introduction

Since the first integrated circuits were constructed, Moore's law has been the primary driver in computer architecture. With improvements in the semiconductor manufacturing process, transistors were getting smaller, faster, and more energy efficient. Dennard scaling also suggested that the power density of transistors would stay the same as transistors become smaller. These scaling laws predicted the technology improvements well, until early this century when Dennard scaling ended [21]. Limitations on power budget and heat dissipation lead to restrictions in operational frequency and the ratio of active logic vs quiescent logic, forcing a shift away from ever more complex and deeply piplined processors.

Despite chip manufacturers having trouble reducing power consumption and increasing performance, the market demand for high-performance and low-power computing, meanwhile, continues to grow. The traditional general-purpose central processing units (CPUs) do not scale to all workloads. Even with the ability to redesign the instruction set architecture (ISA), e.g. RISC-V or AArch64, or with extra instruction extensions such as SIMD, CPUs failed to efficiently exploit massive parallelism from highly parallel workloads such as neural network training. General-purpose GPUs (GPGPUs) became essential for such workloads. Power consumption is another major concern. Technologies such as speech recognition and voice assistant are now ubiquitous, and they are increasingly becoming always-on. For mobile and Internet of things (IoT) devices, processing the data on CPUs or even graphics processing units (GPUs) can be a burden to devices' tight power budget. Other than devices running on their limited battery power, heat dissipation also sets a power budget for high-performance computing scenarios. Venkatesh shows that restrictions on heat dissipation have significantly limited the number of transistors that can toggle simultaneously, a scenario called dark silicon [117]. As a result, we could not simply place more transistors on to a chip to increase its performance.

As a result, we observed a paradigm shift from general-purpose cores to specialised hardware. Custom accelerators, implemented either as application specific integrated circuits (ASICs) or on field programmable gate arrays (FPGAs), can carry out computations orders of magnitude faster and more energy efficient than general-purpose processors. For high-performance computing, major cloud providers, including Amazon Web Services, Google Cloud Platform, and Microsoft Azure, are all providing GPU instances and/or FPGA instances to facilitate computational-heavy but parallelisable workload. Google has also developed its own Tensor Processing Unit (TPU) to accelerate machine learning and deep neural network workloads [69]. This trend is especially visible for mobile computing; a mobile system-on-chip (SoC) today contains many CPU cores with different sizes and specialised for different power/performance trade-offs, a GPU for rendering and video decoding, digtial signal processors (DSPs) for communication and processing sensor data, neural processing units (NPU) for machine-learning-powered features such as photo enhancement and speech recognition and many more specialised accelerators [105] for various tasks.

The paradigm shift, of course, is not free. Accelerators designs have vastly different programming models compared to general userspace code, and the incompatibility means that even porting code to use existing accelerators could be a costly task. For example, traditionally accelerators work with physical addresses while applications operate within the virtual address space. As virtual pages are not guaranteed to be continuous in the physical address space, a common apporach would be to require user-space programs to use a physically-contiguous memory regions, specially allocated by the driver through special routines. Doing so may require a fundamental rewrite of the original userspace program to be accelerated, as the method for managing memory and other resources are completely changed.

This is a pattern that we already observed for GPU programming. GPU application programming interfaces (APIs) exposed to the userspace programs are hard to use; it requires special routines for resource management, buffer allocation, data transfer, etc. For GPU the additional complexity could be justified by the fact that GPUs are already commodity products like CPUs, and these APIs can be used by a wide range of users. However, this situation is unsatisfactory for accelerators and increases the barrier for converting workloads to accelerators.

Another common difference when programming accelerators is the absence of efficient shared memory. Most accelerators today operate on their own private scratchpad memory, and use direct memory access (DMA) to transfer data between the scratchpads and main memory. Their absence from the cache hierarchy means that access to memory locations that are potentially shared between multiple hardware units (e.g. CPUs or other accelerators)

is likely to be very costly in terms of latency and on-chip interconnect bandwidth. This potentially limits the type of tasks that can be offloaded to accelerators to synchronisation-free streaming workloads. Tasks that require fine-grained communication between CPUs and accelerators, e.g. locking, operating shared or concurrent data structures, are problematic due to the high communication overhead.

It would be ideal if we can bridge the programming models, and make programming on heterogeneous systems not much different from existing programs. I believe that one key step is to allow accelerators to operate with virtual addresses and simulatenously participate in the cache hierarchy. Treating accelerators as peripheral devices and communicating based on physical addresses and DMA place tremendous limitations on accelerator design and what applications can do, e.g. requiring the use of continuous, pre-memory-mapped region or requiring a driver to translate addresses for all requests.

Previous work has explored allowing accelerators to operate with virtual addresses [59], and there were separate explorations on empowering accelerators to have coherent caches [105]. However, these efforts do not combine well. Without virtual-address based cache coherence, each such accelerator would need to perform its own address translation. This means that it would need to have a dedicated translation lookaside buffer (TLB) that is large enough to fit its working set and it would need access to hardware for page table walking and memory protection. Furthermore, address translation would be a part of the critical access path for memory access. This might work for an individual accelerator, but does not scale when we move to systems with seas of accelerators and accelerator-rich architectures [30].

1.1 Hypothesis

My hypothesis is that a practical cache coherence protocol can be developed that has virtual addressing, with address translation and protection integrated as the fundamental part of the protocol. Such cache coherence protocol could empower all hosts that access memory, no matter how small, to acquire virtual addressing capabilities, without having to bear the cost of expensive hardware for addressing translation and memory management. The hardware and resources needed for address translation, such as the TLB and the page table walker (PTW), can now be centralised and shared, and no longer on the critical path for L1 cache accesses. Sharing of these infrastructures would also allow use of more advanced components, like complex TLBs, as the cost is amortized by all agents rather than duplicated for each host.

Such a protocol can unlock new categories of accelerators. With easy access to userspace memory, the accelerator can follow programming model of userspace programs more closely;

low overhead of virtual memory access can allow accelerators to overcome the limitation by Amdahl's law [3] by allowing more fragments of a program to be accelerated.

Such a protocol not only can help devices that are traditionally peripheral to gain fast and direct access to virtual memory spaces, but can also be used by the general-purpose CPU. These CPU can offload the memory management tasks from the critical path for accessing the L1 caches, and use simpler virtually-indexed virtually-tagged (VIVT) caches, reducing complexity in the L1 fast path, reduce energy consumption by eliminating TLB accesses and reduce area requirement by removing memory management unit (MMU) hardware near the core.

1.2 Contributions

The primary contribution of this thesis is a virtual cache coherence protocol that permits the use of VIVT L1 caches. The protocol is designed to allow commonly used read-only synonyms to reside in caches while still maintaining correctness in hardware when read-write synonyms occur. The protocol is designed and described in detail, and implemented and evaluated on a FPGA. Caches that communicate using the protocol are implemented, and support has been added to a Linux kernel port. Systems with the protocol have lower resource utilisation and higher maximum frequency compared to the physically coherent counterpart as the TLB is removed from the L1 and the critical path of memory access, while being still comparable in terms of performance per MHz. The flexibility and advantages of the protocol are demonstrated by the creation and integration of easy-to-use accelerators that can be accessed from the general-purpose cores with a low latency. Accelerators that can efficiently directly access userspace virtual memory and perform virtually addressed atomic memory operations are demonstrated.

The process of exploring and creating the protocol required current high-performance simulation techniques to be extended. A high-performance simulator is presented for exploring just the behaviour of TLBs. This is then extended to provide fast cycle-level simulator. The simulator employs an innovative technique to combine binary translation with cycle-level simulation and therefore significantly speedup the simulation process compared to traditional interpretation-based simulators. The simulator built, R2VM, can achieve $\sim$30 million instructions per second (MIPS) in cycle-level simulation in lockstep execution mode, more than 100x the performance of gem5 in a similar mode of operation. For non-cycle-level fast-forward execution, R2VM can achieve $>$400 MIPS per thread. This is significantly faster than gem5's 3 MIPS fast-forward execution, and even better than emulators that exploit

dynamic binary translation (DBT), such as QEMU. This is the first binary translated simulator that supports cycle-level multi-core simulation.

As the basis of the virtual cache implementation, I also developed and open-sourced a collection of open-source processor and SoC components, called Muntjac. Muntjac contains implementation of a RISC-V (RV64GC) core with machine and supervisor privilege levels, as well as cache subsystems and interconnect components that utilise TileLink. An untethered Linux-capable example SoC is implemented with these components. Muntjac core can achieve Dhrystone score of 2.17 DMIPS/MHz and CoreMark score of 3.01 CoreMark/MHz. Muntjac is designed to be modular, verifiable and extendable, and be a good starting point for education, research and industrial applications.

1.3 Thesis Outline

- Chapter 3 discusses in-depth about simulation techniques; these techniques lead to a very fast yet powerful cycle-level simulator. The simulators built are used for various experiments that profile the performance characteristics of software TLB flushes, address space identifier (ASID) space sharing and synonym usage patterns.

- Chapter 4 utilises the fast simulator and experiment results in Chapter 3 to guide the design of a new cache coherence protocol that empowers virtual cache; all issues encountered related to the cache coherence protocol discovered in simulation before any hardware implementation are depicted in this chapter.

- Chapter 5 describes all hardware work related to the cache coherence protocol. The chapter describes the design of Muntjac, an open-source RISC-V processor and relevant components. Muntjac is then used as the baseline for the implementation of the cache coherence protocol discussed in Chapter 3. Since hardware implementation can deviate a lot from the simulator, some issues discovered during the implementation process are described in this chapter. This chapter includes the final implemented design of the cache coherence protocol and designs of caches that use it.

- Chapter 6 describes the application of the virtual cache coherency for accelerators. I compared the performance of physical accelerator, accelerator that uses virtual addressing and other configurations. On top of that, I have designed a completion notification mechanism that bases on top of the virtual cache coherence protocol, and an “accelerator core” which behaves like a device to the operating system (OS) but can execute arbitrary userspace code and perform acceleration.

1.4 Publications

Research from this dissertation has been published at the following workshops, conferences and/or technical reports:

- Xuan Guo and Robert Mullins. Fast TLB Simulation for RISC-V Systems. In *Third Workshop on Computer Architecture Research with RISC-V*, 2019. [52]
- Xuan Guo and Robert Mullins. Accelerate Cycle-Level Full-System Simulation of Multi-Core RISC-V Systems with Binary Translation. In *Fourth Workshop on Computer Architecture Research with RISC-V*, 2020. [53]
- Xuan Guo, Daniel Bates, Robert Mullins, and Alex Bradbury. Muntjac – Open Source Multicore RV64 Linux-capable SoC. In *First Workshop on Open-Source Computer Architecture Research*, 2022. [54]
- Xuan Guo, Daniel Bates, Robert Mullins, and Alex Bradbury. Muntjac multicore RV64 processor: introduction and microarchitectural guide. Technical report, University of Cambridge, 2022. [55]

During my PhD study, I also contributed to research that led to the following publications and presentations that is not included in this dissertation:

- Yiren Zhao[1], Xitong Gao[1], Xuan Guo[1], Junyi Liu, Erwei Wang, Robert Mullins, Peter YK Cheung, George Constantinides, and Cheng-Zhong Xu. Automatic generation of multi-precision multi-arithmetic CNN accelerators for FPGAs. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019. [126]
- Xuan Guo. pin-init: safe initialisation of pinned structs. In *Kangrejos 2022 The Rust for Linux Workshop*, 2022.

[1] Equal contribution

Chapter 2

Background

2.1 Virtual Memory Support for Accelerators

Unified virtual-memory (UVM) is an important way to allow cheaper and easier communication between software and accelerators. Traditionally, accelerators work on physical address space while applications on virtual address space, so the device driver is responsible for translating the addresses used in requests to maintain correctness. As virtual pages are not guaranteed to be continuous in physical address space, we either need to require user-space programs to allocate physically-contiguous memory regions from special routines, or the DMA controllers would need to be scatter-gather capable, and the device driver would translate a single virtually-addressed region to multiple physical regions before feeding them to the DMA engine.

Many existing systems already feature I/O memory management units (IOMMUs). These IOMMUs, however, operate as separate devices, use a different set of page tables, and require dedicated drivers. They are primarily used by hypervisor software to provide IO pass-through support in virtualisation systems, or by the OS as a mechanism to perform access control and protect against potentially malicious DMA devices (e.g. Thunderbolt devices) from accessing arbitrary physical memory.

Cong proposed a IOMMU design for use in accelerator-rich architectures in 2012 [29]. It suggests having IOMMUs as part of accelerators, translating DMA requests instead of memory burst transactions coming out from DMA controllers. On a TLB miss, the operating system is interrupted to help to refill the TLB. Chen refined the design, replacing multiple IOMMUs with a single IOMMU supporting multiple concurrent translations [28]. A new design of the same group is proposed in 2017 [59], utilising private TLBs for each accelerator with also a shared TLB for all accelerators to use. In this new design, they also showed that

by modifying the CPU to use its page table walker to walk pages for the accelerators (instead of calling into the OS), the cost of a TLB miss could be greatly reduced.

Kumar proposed a more general-purpose design by incorporating caches into the design [75]. Each accelerator owns a small slice of private cache, while all accelerators together share a larger cache. The accelerator cache is separate from CPU caches, and it is VIVT, so the IOMMU does not need to be consulted on a cache hit. The accelerator cache and the last-level cache (LLC) is kept coherent by using a TLB and a reverse map to translate addresses between virtual and physical domain. Synonyms are not supported, and only one copy of any given physical address may be present in the virtual cache.

2.2 Cache Coherent Accelerators

I/O coherence is a one-way coherence that ensures memory access by an I/O device (or in this case, DMA accesses performed by accelerators) can retrieve updated copies from CPU caches instead of stale data in the main memory in case of a read, or invalidate corresponding cache lines in CPU L1 caches in case of a write [14, 86]. Memory accesses from the CPU, on the other hand, would not generate bus transactions to the peripheral devices, and do not take possible private caches in the I/O device into account.

Recently, there has been an emerging interest in exploring beyond I/O coherency and allowing peripherals to actually have coherent caches [105]. IBM took the lead in the industry by proposing coherent accelerator processor interface (CAPI) in 2015, which extends the PCIe protocol and allows its POWER8 cores to snoop caches on the FPGA [111]. AMD followed suit and formed the cache coherent interconnect for accelerator (CCIX) consortium with ARM, Xilinx and other companies in 2016 [31]. CCIX never gained much momentum and was succeeded by Compute Express Link (CXL) developed by Intel and later made an open standard in 2019 [106]. Both CCIX and CXL are based on PCIe, similar to CAPI. Accelerators, under all three of these protocols, are still considered peripheral devices and not tightly integrated with CPUs. Furthermore, cache coherence of these protocols is based on physical addresses, and they do not have provisions for supporting virtual memory.

While there are many explorations in adding virtual addressing support to and incorporating caches into accelerators separately, the design is much more difficult if accelerators want to utilise both coherent caches and virtual memory. If the cache is physically tagged, then translating from virtual addresses to physical ones is inevitable for all memory accesses, and therefore TLBs are required near the accelerator, and address translation becomes part of the critical path for memory accesses [43]. With virtual caching, sharing address translation infrastructure would be made possible, but synonyms become a problem.

2.3 Sharing Address Translation Hardware

Address translation and protection play important roles in today's processors, supporting multiprocessing and enforcing security. An important part of the design space is the organisation of TLBs.

TLBs are crucial to system performance due to the penalties associated with misses [10, 12]. However, page table walkers and large or advanced TLBs can be very expensive in terms of area or power. Therefore, instead of having to equip all cores with address translation components, sharing address translation hardware is an attractive direction.

2.3.1 ASID Space

Early uniprocessors, such as Intel's 80386, had a single TLB that was simply flushed on a context switch [33]. Flushing the entire TLB on context switch can be undesirable, especially when TLBs are shared. ASIDs, unique numbers allocated to distinguish distinct address spaces, are therefore introduced to reduce the cost of context switch. TLB entries are tagged with ASIDs, and the tag is checked against current ASID to determine if it should be used. Use of ASIDs can be traced to systems as early as MIPS R2000/R3000 [68].

When ISAs without ASID support adds them as a later extension, e.g. x86 [66], they often keep the ASID space local to hardware threads to keep backward compatibility, because originally each core naturally requires separate address translation units. On the other hand, for architectures that are designed to have ASID support from day one, such as ARMv8, the ASID space is global to all processors, and remote TLB cache shootdown is also supported.

The ASID situation in RISC-V is described later in Section 2.6.2.

2.3.2 Shared TLB

With a global ASID space, remote TLB shootdown, shared last-level TLBs and also inter-core cooperative prefetching [15, 84] become possible. Bhattacharjee, et al. investigated the potential performance gains of using a single shared L2 TLB when compared to multiple per-core private L2 TLBs (with the same number of total entries when aggregated) [16]. Their result showed that shared TLBs can consistently outperform private TLBs, and on average the L2 miss rate is reduced by 27%.

2.4 Virtual Cache

When memory accesses are made using virtual addresses, address translations are inevitable when physical addresses are used. The address translation either has to happen before it is used to index into the cache (e.g. physically-indexed physically-tagged (PIPT) caches), or has to happen in parallel with cache access, and be used to check against tags (e.g. virtually-indexed physically-tagged (VIPT) caches, or PIPT caches with index bits limited to within the page offset). To avoid accesses to address translation data structures, translations are usually cached and stored in a TLB. Therefore, in each memory access, essentially at least two caches are accessed on a cache hit.

A physically-tagged cache coupled with a TLB has been overwhelmingly popular and is used in almost all commercial systems seen today. On the other hand, VIVT caches are rarely seen in general-purpose CPUs. The reason behind the industrial choice is due to the two potential problems, synonyms and homonyms.

Homonyms are the same virtual addresses which represent different physical addresses. For this scenario to happen, the virtual addresses would be translated under different page tables, i.e. different processes. The homonym issue is simple to solve by tagging virtual addresses with ASIDs. The synonym problem, however, is much more difficult to solve.

2.5 Existing Solutions for Synonyms

Synonyms are different virtual addresses which point to the same physical address, arising from memory sharing between different processes (their uses are described in detail later in Section 3.4). If synonyms are not specially dealt with, then it is possible that multiple cache lines in L1 caches may be granted write permission to different virtual addresses but end up referencing the same physical address. Writes to one of them will not be propagated to other copies, causing data inconsistency. Synonyms can both appear within a single cache (referred to later as *intra-cache* synonyms) or span multiple caches (referred to later as *inter-cache* synonyms). Various solutions to the synonym problem have been proposed, which will be discussed in detail in this section.

2.5.1 Synonym Prevention

The simplest solution to the synonym problem is to eliminate the need for synonyms at all. By guaranteeing that no synonyms will be created by the software, it simplifies the design of caches and the cache coherence protocol significantly.

This strategy was adopted in some early processor designs, such as SPUR, the third generation RISC processor from UC Berkeley [63]. In SPUR, caches are virtually-indexed and virtually-tagged. With no L2 caches, it stored cached page table entries (PTEs) and data within the same cache. The address translation only happens on a cache miss. It uses a snooping cache coherence protocol. The protocol carries both virtual and physical addresses. The virtual address is used by other cores snooping on the bus, and the physical address is used to access the main memory. SPUR has a global, segmented virtual address space, where different processes operate under different segments. The authors comment that the synonym problem "is hard to solve in a single virtual-tagged cache, and even harder to solve in a multiprocessor system with many such caches. SPUR avoids this problem by disallowing synonyms. Instead, two or more processes share information by putting it in a shared segment at the same displacement" [63].

A single-address-space operating system [27] eliminates synonyms by letting all processes share a single and global virtual address space. Different processes thus can access the shared pages using the same virtual addresses, synonyms are not required.

Segmented addressing [67] side-steps the synonym issue in a similar way to the single-address-space approach. Instead of a single-phase virtual to physical address translation, it employs a two-phase address translation. In this scheme, process-local virtual addresses are first translated into addresses in a global virtual address space using segments; those addresses are subsequently translated to physical addresses. The first phase translation uses segments, hence the name. Two processes are allowed to share segments, but all global virtual addresses must correspond to unique physical addresses and synonyms are disallowed in this step.

I consider synonym prevention techniques problematic. Section 3.4 shows that various forms of synonyms are widely used in today's operating system. The proposed approaches to prevent synonyms all require large-scale changes to the OSes. They also cause the virtual memory system to change dramatically, so a new ISA would be required.

2.5.2 Synonym Avoidance

An alternative to completely eliminating synonyms in software is for the OS to still use synonyms but ensure that they are not exposed to the hardware. An OS can utilise page tables and page faults to guarantee that no synonyms are visible in hardware, i.e. only one virtual page can be mapped to a given physical page at a time, and all other virtual pages pointing to the physical page be marked as invalid. When a synonym page is accessed, a page fault will trigger, and the OS can unmap the previous translation and map the new one. This operation

is however expensive due to the high cost of flushing caches [24]. For performance reasons, this approach is not in use in modern hardware and OS designs.

2.5.3 Physical Tagging

The two approaches summarised above prevent synonyms from occurring at the hardware level. Due to the usefulness of having synonyms, other approaches try to handle them by detecting their presence in hardware. There are two basic approaches: brute-force search and reverse maps.

Brute-force search essentially searches all possible sets which may contain synonyms during cache misses. Brute-force search can work well with VIPT caches. A VIPT L1 caches tag each line with a corresponding physical address. As typically only a few bits of the virtual page number are used to index the cache, only a few sets in the cache may possibly contain synonym to a given address. When a cache misses, the controller searches in every possible set that may contain synonyms, and invalidates or retags existing cache lines. This approach is used by AMD's second generation Opteron processors [77]. The OS may further aid the cache controller by aligning synonyms properly, so all synonyms fall into the same set [24]. I will not discuss the alignment further here as it essentially converts a VIPT cache into a PIPT cache.

VIPT caches offer the advantage of virtual caches while using physical tags to minimise the impact of the synonym problem, so they are widely employed in commercial designs, especially those without out-of-order (OoO) execution (OoO processors often simply use PIPT caches instead, as their ability to execute out-of-order can usually hide the latency introduced by TLB lookups or smaller caches, e.g. ARM cores frequently use micro-TLBs to translate addresses before cache access [4], while Intel and AMD often size their L1 caches so that only page offset is needed for cache access [58, 112]).

However, physical tagging requires address translation to complete before tag comparison. Therefore, a TLB access for each memory access becomes inevitable. As around 20% of energy consumption of memory access comes from the TLB [70], VIPT caches are more power hungry compared to VIVT ones. Brute-force search does not work well with VIVT caches, because it relies on physical address information, which is missing in VIVT caches.

Zheng's speculatively indexed physically tagged (SIPT) cache [127] is conceptually a hybrid between VIPT and PIPT caches. It tries to predict the address bits beyond page offset using virtual addresses and use it to speculatively index the cache, and then check that the speculation is correct by comparing it against the actual physical address with the result of TLB lookup. It avoids the synonym issue in VIPT cache designs, but TLB accesses are not eliminated and an extra prediction structure is needed.

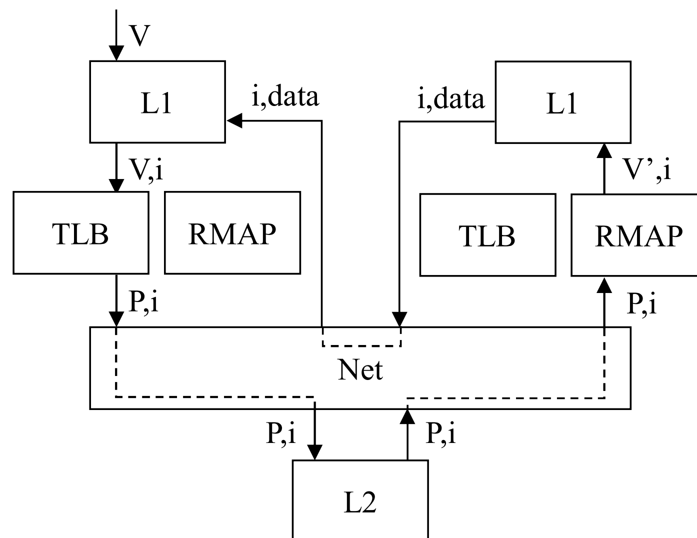


Figure 2.1 Flow in virtual cache system using reverse maps. V, V' are virtual addresses of the same physical address P, and i indicate the transaction id (e.g. MSHR index).

Reverse maps are a hardware structure capable of translating a physical address back to the virtual addresses used in L1 cache tags. Figure 2.1 shows how the ownership of a cache line is transferred between L1 caches in this scheme. During an L1 cache miss, the L2 cache sends out an invalidation message with the physical address. Existing synonyms can be looked up using the reverse mapping, and the previously used virtual address is used to evict the cache line from the L1 cache and complete the invalidation. One important observation is that the reverse mapping must be perfect - i.e. it must provide the correct reverse translation for all entries present in the L1 cache.

One of the earliest solutions using this approach is proposed by Goodman [44]. His proposed solution splits the virtual cache into 3 memory arrays, V-tag memory, R-tag memory and data memory. A normal memory access indexes both V-tag memory and data memory with the virtual address, and compare the tag stored in V-tag memory. The cache controller snoops the physical address on the bus and searches it against its R-tag memory. R-tag memory stores the index to the V-tag and data memory, and therefore the correct cache line can be evicted when necessary. This solution works really well when R-tag memory is fully associative, but when R-tag memory is set-associative, replacement victims must be carefully selected to maintain the perfect reverse translation property. He acknowledges that when

R-tag has small associativity, many cache lines could potentially be wasted, and in extreme cases, it could lead to thrashing.

Kim's U-cache [72] is an optimisation to reverse mapping that addresses the storage capacity limitation of R-tag, by only maintaining information of unaligned pages only (i.e. pages whose lower bits of virtual page number do not match physical page number). U-cache is primarily a solution to the intra-cache synonym issue and does not deal with inter-cache synonyms.

Wang [119] states another approach by storing the reverse mapping information in the cache directory, as opposed to a separate data structure. In this approach, the L2 cache is an inclusive PIPT cache. Apart from the normal tags, metadata stored in an L2 cache line also includes necessary information to pinpoint a cache line entry, e.g. the previously used virtual address or the set number that the cache line is stored in the L1. This information is known as *L2 backpointers*. When an L1 cache misses, the L2 can use this information to invalidate the proper cache line from the L1 that currently poses it. Wang uses this approach in a private L2 cache, but Cekeleov mentions in his survey paper [24, 25] that this can equally apply to a shared L2 cache with some extra complexity. This approach is more practical in modern processors than Goodman's approach as the growing size of L1 makes it less feasible to implement a fully associative reverse tag. This kind of technique is used in MIPS R10000 to ensure correctness in presence of synonyms [121, 87].

Both approaches, however, do not permit a read-only cache line to exist under multiple synonyms (neither intra-cache synonyms within a single L1, nor inter-cache synonyms that span multiple L1 caches). Synonyms are either evicted or re-tagged when accessed with different virtual addresses. This is understandable because the design could be very complex if a reverse lookup may produce multiple virtual addresses for a single physical address, but performance-wise it suffers.

2.5.5 Private Physical L2

Having a private physical L2 cache can mitigate the synonym problem. It is first purposed by Wang [119]. A private physical L2 cache still requires reverse mappings or backpointers to handle synonym issues within the same L1 virtual cache, and therefore still disallows intra-cache synonyms. Synonyms that span across different cores are however allowed, since the inter-core cache coherency is operated using physical addresses only. Disallowing intra-cache synonyms is less a performance overhead compared to disallowing inter-cache synonyms, because usually there are no synonyms within the same process.

The inability to share a L2 cache, however, means that extra area is required for each core. Each core must also still retain a TLB and a page-table walker, even though they are only

accessed on L1 cache misses. Although private L2 caches are common arrangement in commercial high-performance processor designs, they are not ideal for more resource-constrained systems. For example, in a many-core system or a system with seas of accelerators, the cost of private L2 caches and the retained MMU infrastructure can be expensive and undesirable if each core/accelerator requires a copy of them. Forcing the use of private L2 caches also means that the core-to-core latency will inevitably increase, as each synchronisation must traverse through more levels of the cache hierarchy.

2.5.6 Synonym Remapping

It is also possible to deal with synonyms before they enter the cache hierarchy; such techniques are known collectively as synonym remapping. Synonym remapping tries to resolve virtual addresses with synonyms to a globally unique address before accessing the cache.

Qiu proposed a scheme that maps synonym virtual addresses to a unique primary virtual address [100]. In Qiu's scheme, in the case of a synonym it is the OS's responsibility to select one of the virtual addresses as primary address and others as secondary virtual addresses in case of synonym. A "synonym-lookaside buffer" (SLB) is introduced to hold mappings from secondary addresses to their primary address, and the OS is required to maintain the synonym relations and insert a mapping into the SLB whenever necessary. All memory accesses will check the SLB to see if any entry hits. If there is a SLB hit, then the entry provides a new address to use for cache access. If no entries hit, it is assumed that the address is either a primary virtual address itself or there are no synonyms, therefore the original address is used. In essence, SLB ensures that all memory locations are accessed using the primary virtual address only by redirecting all secondary virtual addresses to their corresponding primary virtual addresses. Because the SLB is completely software maintained, the scheme also requires the OS to record whether an address is secondary or not is recorded in the page table. When an address is neither present in the SLB or in the cache, the TLB is able to detect whether the access is to a secondary address using the information in the PTE. If it is, the TLB will generate a trap and the OS can refill the SLB. Compared to a TLB, an SLB is simpler and consumes less energy. The control logic is also simpler because no page walking is needed when the SLB misses.

Yoon's dynamic synonym remapping takes the idea further by freeing the management task from the OS [122]. Instead of having the OS defining which address is primary, Yoon's scheme instead defines the first virtual address used to access a particular physical address as primary (Yoon calls this *leading* virtual address). His design performs active synonym detection by adding an extra table between the L1 virtual cache and the L2 physical cache. When a synonym is detected, a new entry is added to the address remapping table. It further

reduces the access to the remapping table by introducing a synonym signature (implemented using a Bloom filter), which filters out the accesses that are known to not have synonyms. Yoon claimed that only 2% of all requests require the remapping table to be accessed, therefore the approach saves more power compared to Qiu's SLB.

Yoon also uses a similar primary virtual address idea in his later work [123], but moves the bulky remapping structure from before the L1 cache access to after the L1 cache access. The scheme described in this paper forces a cache miss if the virtual address accessed is not a primary virtual address. After missing the L1, the primary virtual address is found and the access request is replayed using the primary address. This will cause significant slowdowns for synonym accesses. Yoon claimed that this would not cause overall performance issues for GPU access patterns, which is the focus of this work.

Overall these techniques resolve the synonym before cache accesses, thus they require no changes to cache coherence protocols. However, Qiu's scheme requires extensive OS modification and Yoon's first scheme requires complex per-core structures. Yoon's second scheme removes the need for per-core structures, but it still needs a shared remapping structure and will result in performance degradation for synonym accesses.

2.5.7 Hybrid Caching

Hybrid caches are caches that can operate both as physical caches and as virtual caches. Usually, the physical/virtual granularity is at the cache line level, so it may contain a hybrid of physical and virtual cache line entries.

Opportunistic virtual caching (OVC) proposed by Basu [11] lets the operating system select whether virtual addresses should be used to address the L1 cache on a per-page basis. In many cases where there is no conflicting use of synonyms (i.e. read-only synonyms), the OS can tell the hardware to access the L1 cache using virtual addresses. In applications where synonyms are actually used, the OS can instead opt-out, and still access the cache using physical addresses. Basu's scheme uses the most significant bit of the virtual address to indicate which mode the L1 cache should be accessed in, so there is no hardware lookup required to determine whether the TLB should be used. This scheme effectively gates the TLB at the discretion of the operating system.

The scheme proposed by Park [91] also employs hybrid caching, but it uses a synonym filter instead of a single bit from the address. All accesses go through the synonym filter. If the filter hits, then the TLB is accessed and the cache is accessed using the physical address. If the filter misses, then the cache is accessed using the virtual address instead. It requires minimal changes to the cache (merely an additional bit to distinguish between physical and virtual) and the TLB. The filter is a Bloom filter [19] managed by the OS. The filter must be

updated when page mappings are changed, which unfortunately can be very expensive during unmapping, as entries could not be removed from the Bloom filter. In addition, care must be taken when adding new entries to the Bloom filter, because some addresses previously accessed by virtual addresses might be accessed by physical addresses now as the Bloom filter may produce false positives.

Devirtualised memory proposed by Haria attempts to map virtual pages to identically addressed physical pages whenever possible, while still allowing non-identical page mapping when needed. In this scheme, a bitmap is used to indicate whether a page is identity-mapped and the permission bits of the page if it is. For pages which are not identity-mapped, conventional address translation is performed to generate the translated physical address [60]. By requiring synonyms pages (or pages used by external accelerators) to be identity-mapped, the synonym issue is mitigated. This approach, however, requires extensive OS support and seriously reduces the flexibility of the virtual memory system. Haria also acknowledges that the scheme may be more vulnerable to Meltdown [79] and Spectre [74] attacks because the physical address is essentially exposed to the userspace. This technique is strictly not hybrid caching as the cache is always accessed via virtual address (but PA = VA), but in the spirit, it addresses synonyms by using physical addresses, so I classify it under this section.

Hybrid caching saves power consumption significantly by gating TLB accesses all the time. However, it fails to remove TLBs and page table walkers completely, as those structures are still accessed occasionally.

2.5.8 Intermediate Address Space

One way to gain many of the performance benefits of virtual caches without dealing with the synonym issue is to use an intermediate address space. Midgard [56] is an approach that uses this idea. Instead of translating virtual addresses directly to the physical address space, Gupta proposed that a Midgard address space is added as another level of indirection. There is a single Midgard address space shared by the system; Virtual memory areas (VMAs) of different processes are mapped into unique addresses in this unified Midgard address space. Pages in the Midgard address space are then translated to physical addresses using conventional page tables.

VMA is a Linux concept used to manage virtual addresses in a more coarse-grained manner than page-based MMU [46]. While a single `mmap` system call may cover multiple pages, the OS can group them into a single VMA for management. The authors claim that because there are far fewer VMAs, using them for the first-stage translation reduces the chance of translation cache misses.

The approach is not without drawbacks, however. First of all, because VMAs are ranges, checking an address against a VMA entry requires two bound comparisons and an addition. This means that it is intrinsically more complicated and much slower when implemented in hardware. The authors mitigated this issue with an additional page-based TLB (referred to as L1 VLB), which is used before the VMA-based buffer (L2 VLB) is checked. The additional levels further increase the area cost of this design. The range-based nature also means that the buffer for virtual address to Midgard address translation must be fully associative, increasing area, implementation complexity and power consumption.

Furthermore, one important assumption made in the Midgard design was that pages from different VMAs cannot alias (otherwise there will be synonym issues in the Midgard-to-physical mapping). While this assumption frequently holds, it is possible for two pages in different VMAs to refer to the same physical page if kernel same-page merging is enabled [71]. The feature is designed to merge pages (of different VMAs) of identical contents to the same copy-on-write physical page. While this feature is an opt-in feature and rarely used by normal applications, it is frequently used by virtual machine applications to reduce memory usage. If pages from different VMAs may alias, it would be necessary to find a solution to the synonym issue at the Midgard address level.

2.5.9 Write-Through

Kaxiras and Ros state that the reverse maps used in virtual-address-based cache coherence protocol could be omitted completely if all requests flow strictly one-way from the virtual address domain to the physical address domain, and never backward [70]. Invalidation messages used in MSI-like protocols violate this condition. This, however, is not an issue with write-through caches.

Write-through caches are frequently employed in GPU-like devices. The use of write-through caches in a multi-core or many-core CPU system is problematic, as it both generates a large amount of bus traffic and increases the latency of write operations.

They described a cache coherence protocol called VIPS. It tackles the challenge of increased bus traffic using two methods: first, the OS is required to tag whether a page may be shared between processors. If a page is strictly private to a core, then write-back could be used, as there is no need for coherency. Second, write-through does not happen immediately but is delayed. When a cache line transitions from clean to dirty, a counter associated with the cache line is started. When the counter counts down to zero, the cache line will be written back. If a cache line is already dirty then further writes will not reset the timer. To ensure coherence, all shared clean cache lines are invalidated when a fence is encountered, and all dirty cache lines are written back in time order.

One problem with this scheme is that it requires the OS to classify pages as shared or private. However, many access patterns are dynamic and it is hard for the OS to know ahead of time. For multi-threaded applications, the OS might need to conservatively mark all pages as shared (thus write-through). They addressed this problem in their later work by classifying pages in hardware [103, 40].

There are a few other problems with this approach in my opinion. First, it requires a weak memory model. In fact, the cache coherence protocol described in the paper [70] does not even address how to cope with atomics and load-reserved/store-conditional (LR/SC) patterns in the protocol. Atomic instructions are vital for a simultaneous multiprocessing (SMP) operating system, such as Linux. Second, the paper does not consider the possible effect of operating systems when claiming synchronisation is rare. As most OS structures are guarded by locks, requiring a whole cache flush when acquiring the lock might be problematic. Finally, the rare synchronisation assumption is becoming less valid with the rise of modern programming languages and patterns. Fine-grained locks are gaining popularity [98], especially in memory-safe languages like Rust. Making the situation even less favourable, multi-threaded languages that use automatic reference counting like Swift [64] require the use of atomics for each retain (increase the counter) and release (decrease the counter) operation, so synchronisation can be common, if not frequent.

2.6 RISC-V

RISC-V is a simple, extensible, general-purpose and open ISA. Since the beginning of my PhD study, RISC-V support has been upstreamed in many open-source software projects, such as the Linux kernel, GNU toolchains, LLVM, etc. We also see an emerging number of open-source RISC-V processors, such as Ariane [124] from ETH Zurich, and SweRV from Western Digital Corporation. There are also a number of open-source SoC platforms being actively developed, such as OpenTitan [83] from lowRISC CIC. With the ongoing ecosystem development of RISC-V and an increasing number of companies and institutions switching to RISC-V for both production and research, RISC-V has become the test bed instruction set of computer architecture research. It is therefore natural that RISC-V is chosen whenever an architecture choice is needed.

2.6.1 Virtual Memory Addressing in RISC-V

The privileged specification 1.10 [120] of RISC-V defines the address translation structure of in-memory page tables to support virtual memory addressing. Depending on machine

word-length and configuration, virtual addresses are translated to physical addresses using two, three or four levels of page table. A control and status register (CSR) called SATP (Supervisor Address Translation and Protection) is used to store the base address of the root page table. In addition to the base address, a paging mode field and a 16-bit ASID field (9-bit in 32-bit RISC-V) are packed into the SATP CSR allowing these fields to be changed atomically. The ASID field can be used to distinguish between TLB entries to avoid the need to flush TLBs during context switches.

RISC-V also includes a `SFENCE.VMA` instruction to flush TLBs. `SFENCE.VMA` takes two optional register operands to specify the ASID and virtual address to flush. Implementations may choose to ignore register operands and always perform a full TLB flush.

Unlike almost all other architectures which describe similar instructions for flushing translation cache entries from TLBs, RISC-V defines the instruction as a fencing instruction after which the previous modifications to address translation data structures are required to be honoured by the hardware. The specification is worded in such way deliberately to allow a wide design space. However, as a result of such definition, an implementation can technically cache invalid entries in its TLB while still being RISC-V compliant.

2.6.2 ASID Space

In 2017, at the beginning of my PhD study, RISC-V mandates that all hardware threads (harts) need to have their own private ASID spaces. This is partly due to legacy hardware and software. The RISC-V privileged specification was undergoing some large-scale changes at the time, and there was already software that made use of the virtual memory system without having ASID support. As RISC-V requires CSR fields unknown to software to be filled with 0, it happens that software naturally uses ASID 0 when ASIDs are introduced. The backward compatibility burden essentially requires ASID 0 to be treated as local.

Following my suggestions about a global ASID space, the RISC-V's privileged specification committee has decided to place a forward-compatibility requirement on software so that non-zero ASIDs should have consistent meanings across different harts, and software should not use ASID 0 if it decides to use other ASIDs. I evaluated the impact of this decision on the potential effects of shared TLB designs in Section 3.3.1. The result leads to a proposed extension which allows RISC-V hardware to provide a hint to software as whether the ASID space is local, global or partially shared [52].

In 2019, Linux had not yet made use of ASIDs in RISC-V. With the help of TLBSim (discussed in Section 3.1 and Section 3.2.1), I have developed a patch series that makes use of ASIDs for the Linux kernel, but the patch series did not make it into the mainline at that time, due to lack of hardware that supports ASIDs and a conflict by another kernel contributor,

Anup Patel, that submits a patch series in the same area. The patch series were combined by Anup Patel and merged into Linux kernel in 2021 [92].

2.7 Instruction Set Simulators

Instruction set simulators can be classified as either execution-driven, emulation-driven or trace-driven [23]. We omit a detailed discussion of execution-driven simulators such as Cachegrind [89] that modify programs with binary instrumentation and execute them natively, because they require the host and the guest ISA to be identical and do not support full-system simulation. Emulation-driven simulators emulate the program execution, and gather performance metrics on-the-fly; in contrast, trace-driven simulators run emulation before-hand and gather traces from the program, e.g. branches or memory accesses, and later replay the trace against a specific model. Traces allow ideas to be evaluated quickly without the need to simulate in detail, but cannot easily capture effects that may alter the instructions that are executed, e.g. inter-core interactions or speculative execution [23]. Moreover, storage space required for traces grows linearly with the length of execution, making trace-driven simulators incapable of simulating large benchmarks.

Simulators can also be categorised by their levels of abstraction. One category of simulators is functional simulators. Functional simulators simulate the effects of instructions without taking microarchitectural details into account. Because less information is needed, aggressive optimisations can be performed, and the performance is usually several magnitudes faster than timing simulators. QEMU falls into this category. It should be noted though that while QEMU itself is a purely functional simulator, it can be modified to collect metadata for off-line or on-line cache simulation [116].

The other category is timing simulators. For example, register-transfer level (RTL) simulators can model processor microarchitectures very precisely, but it is not suitable for early experimentation and prototyping as implementing a feature in an RTL simulator is not much different from implementing it in hardware directly. RTL simulators are also poor in performance, usually running at a magnitude of kIPS [113].

At a higher level, there are cycle-level microarchitectural simulators. These are able to omit RTL implementation details to improve performance while retaining a detailed microarchitectural model. A popular example is the gem5 simulator running with In-Order or O3 mode [18]. For faster performance, we can give up some extra microarchitectural details and predict the number of cycles taken for each non-memory instruction instead of computing them in real-time, and in the extreme case, assume all non-memory operation only takes 1 cycle to execute as gem5's "timing simple" CPU model assumes. This approach

is no longer cycle-accurate, but this cycle-approximate model is often adequate to perform cache and memory simulations.

2.7.1 Binary Translation

Binary translation is a technique that accelerates ISA simulation or program instrumentation [62]. An interpreter will fetch, decode and execute the instruction pointed to by the current program counter (PC) one-by-one, while binary translation will, either ahead of time (static binary translation) or in the runtime, i.e. when the block of code if first executed (DBT), translate one or more basic blocks from the simulated ISA to the host's native code, cache the result, and use the translation result next time the same block is executed.

QEMU uses binary translation for cross-ISA simulation or when there is no hardware virtualisation support [13]. On the timing simulation front, Böhm et al. introduced binary translation to single-core timing simulation in 2010 [20].

2.7.2 Multi-core Simulation

Extending single-core simulators to handle multiple cores is complicated by the performance implications of the ways in which cores may interact. As cores share caches and memory, simulations of individual cores cannot simply be run independently. For example, accurate modelling of cache coherency, atomic memory operations and inter-processor interrupts (IPIs) must be considered.

Böhm et al.'s modified ARCSim simulator [20] can model single-core processors with high accuracy and reasonable performance; however, Almer et al.'s extension to Böhm et al.'s work [2] that essentially runs multiple copies of the single-core simulator in parallel threads to provide multi-core support is limited in its fidelity. The author comments "detailed behaviour of the shared second-level cache, processor interconnect and external memory of the simulated multi-core platform" cannot be modelled accurately. QEMU is able to exploit multiple cores to emulate a multi-core guest but provides only a functional simulation mode and supports no timing or modelling of the memory system.

An accurate model of cache coherency and the memory hierarchy requires that multiple cores are simulated in lockstep (or in a way that guarantees equivalent results). Simulators that forego this are unable to properly simulate race conditions and shared resources. Existing cycle-level simulators such as gem5 achieve lockstep by iterating through all simulated cores each cycle. This causes a significant performance drop. Spike (or `riscv-isa-sim`), an interpreter-based RISC-V ISA simulator, supports multi-core simulation by switching between each simulated core in a coarse-grained fashion. Its default compilation option

only switches to the next core every 1000 cycles. The less frequent active core switching makes it impossible to model race conditions where all cores are trying to acquire a lock simultaneously. No existing binary translated simulators can model multi-core interaction in lockstep, and therefore none of these can model cache coherency or shared second-level cache properly.

2.8 Benchmarks

I used the Princeton Application Repository for Shared-Memory Computers (PARSEC) benchmark suite [17] for many performance evaluations throughout this dissertation.

All benchmarks in this suite are written in C or C++, and unfortunately they exhibit a wide range of undefined behaviours. With more recent version of the GCC compiler being more powerful in performing program analyses and optimisations, many bugs surfaced from the poorly written codebase. The benchmark suite is also unmaintained since 2010 so these bugs are not getting fixed.

There are 14 benchmarks in total. Below is a summary of the issues and any changes or fixes that were made.

1. Blackscholes: No issues found.
2. Bodytrack: The codebase uses C++ dynamic exception specifications (`throw(...)`) which are removed in C++11. These cause issues since new GCC versions default to C++11. I have replaced them with C++11-style `noexcept` specifications.
3. Canneal: The codebase uses custom atomic implementations, and RISC-V is unsupported. I have ported it to RISC-V.
4. Dedup: A variable is defined twice and causes linking issues. I made them `static`.
5. Facesim: The codebases rely on wrapping pointer arithmetic which is undefined behaviour. I determined that removing undefined behaviour would require a full rewrite and decided to add the compiler switch `-fwrapv-pointer` to work around this issue. This simply instructs the compiler to assume that pointer arithmetic overflow wraps around using two's complement representation, at the cost of invalidating some optimisations.
6. Ferret: No issues found.
7. Fluidanimate: No issues found.

8. Freqmine: The codebase contains misaligned memory accesses, uninitialized variables and other undefined behaviours. Initial fixes were successful and I used this benchmark for some early experiments, but with further GCC updates bugs returned. A `-fsanitize` run showed more undefined behaviours, and I determined that this benchmark is beyond repair and therefore excluded it from later experiments.

9. Raytrace: Failed to compile; I therefore dropped it from the benchmark suite.

10. Streamcluster: No issues found.

11. Swaptions: No issues found.

12. VIPS: All other PARSEC benchmarks call hooks to denote the start and the end of region of interests (ROIs), but hooks are not called for VIPS. I added the missing calls at relevant locations.

13. X264: Misaligned accesses were detected. I discovered that a newer version of X264 has this issue fixed, and therefore bumped the version from R2970 to R2984.

Many applications in the benchmark suite also have some common compilation issues related to old version of `config.guess` and `config.sub` files that does not recognize RISC-V. I have fixed these issues by updating the files to the latest version. Some of them has additional issues that are fixed by rerunning `autoconf` and `automake` tools.

Chapter 3

High-performance Simulation Techniques for Memory System Exploration

3.1 TLB Simulation

To explore a better virtual memory system for multi-core systems and more heterogeneous architecture, it is important to gather the statistics regarding its usage in today's system. TLB, being a vital part of the virtual memory system, is the centre of this investigation. To explore the trade-offs in TLB designs and how the effect of sharing TLB like some designs do in virtual cache system, I need a fast TLB simulator that can be used to simulate real-life workloads.

Unlike the cache hierarchy, there is no coherency required for TLBs. This means that cycle inaccuracy is more tolerable for TLB simulation. Instead of using a slow timing simulator, I implemented a TLB simulation framework, TLBSim, which can be plugged into faster functional simulators like QEMU to allow TLB simulation while running very large workloads. Several techniques are used to gather useful statistics while not influencing simulation performance significantly. The source code of TLBSim is available on GitHub [48].

3.1.1 Simulator Design

L0 TLB and Inclusive L1 TLB

Because TLBSim relies on existing functional simulators, it is important to understand how they handle virtual memory. When running code of a non-native ISA, QEMU uses its tiny code generator (TCG) to perform DBT, translating guest binaries to host binaries. When

TCG is compiled with soft MMU support (i.e. doing full system emulation), it creates a direct-mapped TLB-like structure in software to accelerate code with virtual memory access. Any guest memory access instructions are translated to a sequence of host instructions which access QEMU's TLB-like structure first to acquire the physical address before performing the actual access. When the TLB access hits, the control flow does not leave generated code, and QEMU's helper code is invoked when the corresponding page does not exist in the TLB. It is worth mentioning that QEMU's direct-mapped TLB is only an acceleration mechanism, and it does not simulate or intend to simulate the actual address translation mechanisms employed in actual hardware. Other ISA simulators, e.g. Spike, use a similar approach to accelerate memory access when soft MMU is used.

The goal of TLBSim is to be easily pluggable into existing ISA simulators without extensive modification. The case is especially true for QEMU, as modifying the TLB used in TCG requires a range of components to be changed. Moreover, functional simulators almost always use direct-mapped TLBs for performance concerns, as the TLB lookup code is extremely hot, and for DBT-capable simulators like QEMU they are generated as inline machine code. Traversing through associative caches, which most hardware TLB designs use, is clearly not acceptable on the hot path. I made the design decision that the code should only be executed on the slow path. When a memory location is accessed, the ISA simulator will query its own TLBs first (referred to later as the L0 TLB from the simulation's point-of-view). If the L0 TLB hits, the simulator can serve the request directly. The TLB simulation framework is called when the L0 TLB misses, replacing ISA simulator's own page walking routine.

To ensure all TLB misses can be accurately recorded and simulated, we need to avoid scenarios where a memory access hits the L0 TLB but would miss in the simulated TLB. To address this potential issue, I enforce that all entries in the L0 TLB must also be included in the simulated L1 TLB. If any entries are evicted from the simulated TLB, the L0 TLB invalidation callback provided by the ISA simulator will be used to remove the entry from the L0 TLB as well.

This approach keeps ISA simulator's fast-path almost untouched, therefore all the performance overhead of TLB simulation lies in the slow path. As a result, only minor performance overhead is observed. A drawback of this approach is that as not all TLB accesses are intercepted, we cannot implement a least-recently used (LRU) replacement policy in L1 TLBs. This is a deliberate trade-off I made for high performance. Other replacement policies that do not require all TLB accesses to be accounted, e.g. FIFO, random or even victim caches, can be supported. I believe that the simulated result is still representative as research suggests that only a small hit-rate gap exists between replacement policies [88].

Thread Safety

An important design decision is also to ensure that TLBSim is thread-safe. This would make it compatible with multi-threaded ISA simulators, e.g. QEMU running in multi-thread TCG (MTTCG) mode, where all guest cores run in their own threads.

To ensure maximum parallelism and minimum runtime performance impact, fine-grained spinlocks are used instead of mutexes where possible. For example, for a set-associative TLB, each set has its own lock for synchronisation purposes. Statistics are gathered using counters implemented as atomic variables.

3.1.2 Simulator Implementation

TLB Models

The TLB models share a common interface, defined by the following C++ abstract class shown in Listing 3.1.

```
class TLB {
public:
    /* ... */

    // Find an entry, and acquire a (possibly) fine-grained lock that
    ↪ prevents
    // any race to the entry.
    virtual bool find_and_lock(tlb_entry_t &entry);

    // Release a (possibly) find-grained lock for an entry.
    virtual void unlock(const tlb_entry_t &entry);

    // Insert an entry, and unlock.
    virtual void insert_and_unlock(const tlb_entry_t &entry);

    virtual void flush_local(asid_t asid, uint64_t vpn);

    /* ... */
};
```

Listing 3.1 Abstract class shared by TLB components in TLBSim

The thread-safety requirement does make the implementation more complex; a TLB can be (transitively) shared by multiple cores and thus accessed in parallel in two threads. Data races or ABA problems can arise if one of the accesses misses and requires a TLB refill.

I have implemented fully associative, set associative and direct mapped TLBs with FIFO replacement policy. Other replacement policies and victim caches can also be incorporated within this framework; the framework permits advanced techniques such as prefetching and CoLT [97], but they are not implemented, being orthogonal to TLB sharing and synonym handling, which are the main topics of this investigation.

Besides the conventional TLB designs used in hardware, I also implement an "ideal TLB", which caches all translations that it sees and never evicts entries. It is later used to implement a software validation tool to validate correct usage of ASID in the OS. It also serves as the basis of the synonym experiment described later.

The framework is able to gather various statistics related to the virtual memory system, including the number of misses, evictions and flushes associated with each level of TLB.

QEMU Integration

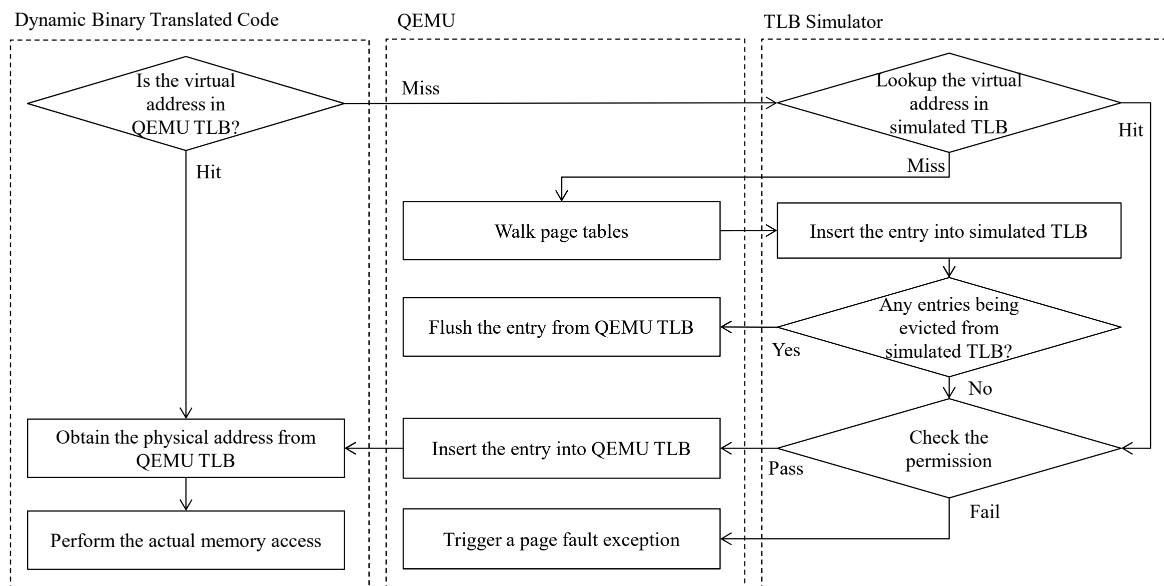


Figure 3.1 Control flow of a guest data memory access in TLBSim, with only L1 caches simulated

Figure 3.1 shows the control flow of a guest data memory accesses when the user is QEMU, with only L1 caches simulated. The control flow is more complicated if multi-level TLBs are simulated.

The design decisions make integration with QEMU simple. To be able to use my TLB simulation framework, only the following changes are made in QEMU:

- Replace page walking routine with calls to the TLB simulator;

- When `SFENCE.VMA` is executed, call the TLB simulator in addition to normal flush;
- Inserting code to the beginning of each DBT-ed basic block to update instruction retirement counters. We need two counters, the total number of instructions executed and the number of memory access instructions executed. By tracking these two counters, we can calculate number of L1 hits even when some of the lookups never reach my TLB simulator.

  To avoid the synchronisation cost by keeping per-CPUs counters, the inserted code will only update per-CPUs counters, and the counters are only aggregated when control is handed back to QEMU's main loop from DBT-ed code.

Apart from running online simulations with QEMU, I have also implemented an offline simulation capability. A trace collector can be connected to L1 TLB models, and log TLB requests to a file. The logged trace file then can be replayed later for different L2 TLB configurations. Due to the amount of data, the offline simulation works well only on small benchmarks; therefore the experiments carried out in later sections used online simulation only.

3.1.3 Simulator Performance

I evaluated the performance overhead of TLB simulation when used together with QEMU. The performance impact is minor, as the only changes I have made to QEMU's fast path are the instruction retirement counters, which can also be turned off if these measurements are not necessary for a specific experiment or benchmark.

Figure 3.2 shows the performance of QEMU with and without the TLB simulation, when simulating a multi-level TLB design when the workload has an L1 TLB miss rate of 1%. When running with 8 guest CPUs cores, the framework achieves around 50 MIPS per core. The average performance loss compared to unmodified QEMU is 18%. In some experiments where the L1 TLB miss rate is low, the performance loss can be as low as 2.5%.

The performance achieved is 2860x faster than the gem5 simulator at 175 KIPS and 125,000x faster than the Chisel C++ RTL simulator at 4 KIPS [113]. The numbers show that if only the virtual memory system is of concern, using my TLB simulation framework provides a significant speedup.

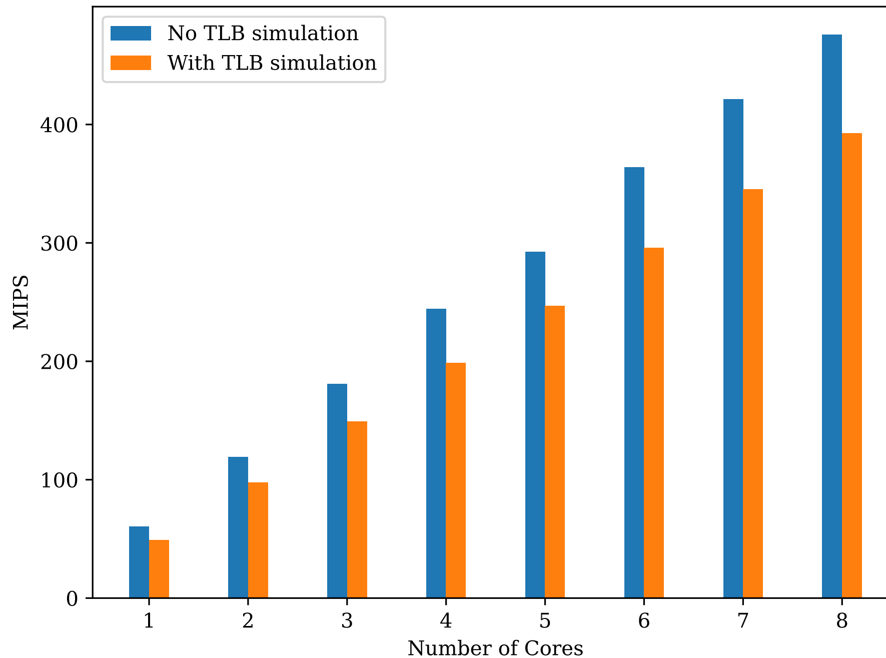


Figure 3.2 Performance of QEMU, with and without TLB simulation, in MIPS

3.2 Software Validation and Profiling

3.2.1 Software Validation

I have used TLBSim to verify OS kernel's behaviour on different hardware implementations. The framework can be configured to disable hardware accessed/dirty bit updates, deliberately cache invalid entries in the TLB (as discussed in Section 2.6.1), and/or generate page faults aggressively when an invalid translation entry is found in the TLB, to test if the software has handled such cases properly. I also implemented a set of runtime assertions on top of the ideal TLB. The following common OS implementation issues can be detected:

- ASID reused without flushing;
- A PTE is updated with a different physical address or a reduced permission, but there is no `SFENCE.VMA` issued.
- Multiple ASIDs used to refer to the same page table. While not strictly wrong, it often indicates an implementation error in software, and is at least a performance issue.

The experiments carried out in Section 3.3.1 require the operating system to have ASID support. I need to implement ASID support for Linux myself, as its RISC-V port does not

have ASID support at the time of work. I used the validation feature of the TLB simulator extensively to verify and debug my implementation.

3.2.2 Software Profiling

As discussed in Section 2.6.1, RISC-V compliant implementations are allowed to cache an invalid entry in its TLB, as `SFENCE.VMA` is specified as an ordering instruction, so it is needed to guarantee that a write to a previously invalid leaf PTE is made visible.

This is contrary to what software normally assumes. Linux, for example, assumes that TLBs cannot contain totally invalid entries. TLB flush routines will not be called by the kernel at all when pages are mapped at previously invalid addresses (e.g. when `mmapping` a new file). The current RISC-V port of Linux (at the time of writing, 5.0) instead uses a feature that Linux provides to allow architectures to preload the MMU (`update_mmu_cache`) to perform the flush instead. This MMU preloading code is called whenever a mapping is created, for example, when a program's text segment is mapped by the `execve` system call. At this time the program is not executed yet, and it is extremely unlikely that a hardware implementation will load these TLB entries speculatively. The TLB flushes have thus most likely been superfluous.

I am interested in how RISC-V's current ISA design can impact software, as TLB flushes can be expensive operations on many micro-architectures, including but not limited to:

- Micro-architectures that always perform a full TLB flush on `SFENCE.VMA`.
- Micro-architectures with multi-level TLBs, in which case a `SFENCE.VMA` requires flushes in all TLB hierarchies.
- Micro-architectures with virtual cache, in which case a `SFENCE.VMA` may also require a cache flush.

Category	Number per MInst	Percentage
Never Accessed	5.01	40%
Previously Invalid	6.78	54%
Previously Non-Writable	0.59	5%
Necessary	0.13	1%

Table 3.1 `SFENCE.VMA` instructions issued by the Linux kernel

I used the simulation framework to investigate `SFENCE.VMA` instructions issued by the Linux kernel. The workload chosen is the compilation of the Linux kernel, as TLB

flushes happen mostly when processes are frequently spawned and terminated. I categorise `SFENCE.VMA` instructions issued into 4 categories, as shown in Table 3.1:

1. Never accessed page: These are mappings that are newly created and never accessed. The specification allows hardware to prefetch TLB entries anywhere in the address space, but these entries will never be accessed in practice.
2. Previously invalid page: These are mappings which were previously invalid, and are made valid by the page fault handler. This usually applies to the stack and heap where page maps are created but memory is not yet allocated. These flushes are not necessary in a design that does not cache invalid entries.
3. Previously non-writable page: These pages are previously non-writable, and are made writable by the page fault handler. These pages are usually results of a `fork` which makes page copy-on-write (CoW). These entries may have been cached in TLBs as the result of a successful read, so it is reasonable to issue `SFENCE.VMA` in this scenario.
4. Necessary flushes: These flushes are absolutely necessary, for example, as a result of `mprotect` which downgrades the permissions on the page.

All cases other than (4) can be done lazily, i.e. do not flush them until they trigger exceptions. The current Linux kernel issues `SFENCE.VMA` instructions in all cases, including (1) and (2), based on the reasoning that TLB flushes are less expensive than page faults. However, in practice, conditions such as (1) and (2) would be rare, if not impossible, to occur in real hardware. As they account for 94% of `SFENCE.VMA` instructions, eliminating them can improve performance when flushes are expensive (which will be the case if a shared TLB is used).

To reduce the number of flushes, what could be done is to optimistically assume that hardware will not cache invalid entries in the TLB. If in rare cases where the hardware does cache these entries, the page fault handler could detect such spurious page faults and fix the situation by issuing a lazy `SFENCE.VMA`.

3.3 Shared TLB and Global ASID

3.3.1 Shared TLB

I envision the architecture suitable for many-core and accelerator systems would need shared address translation services. Requiring each memory host agent (i.e. CPU cores, GPU cores, DMA-capable devices and accelerators) to be equipped with TLBs and PTWs are costly.

Sharing TLB and PTW could be potentially beneficial for such designs. A shared TLB could be equally favourable for typical multi-core systems as well. TLBSim was used to investigate the performance of a shared TLB in multi-core systems, with workloads running on Linux, so that the effects of both the application and the OS are observed.

A shared TLB can be most efficiently implemented if the ASID space is shared globally. As mentioned in the Section 2.6.2, the RISC-V specification mandates that all harts have their own private ASID spaces, ruling out designs that exploit global ASIDs. I used TLBSim to simulate the performance of different designs and understand the potential benefits of a global ASID space. The experimental results are then used to propose an extension.

In this section, I will explore three possible L2 TLB designs and their performance in terms of hit rates. In all three setups, all configurations other than the L2 TLB are kept identical. All simulations are carried out using 8 cores, and each core has its own separate L1 instruction TLB and L1 data TLB. All L1 TLBs are 32-entry and fully-associative. Hardware accessed/dirty bit updates are on, and no invalid entries will be cached. A write access to a cached read-only TLB entry may trigger a page fault directly without falling through to the PTW (even if the underlying PTE has already been changed to writable at the time of access). The three setups differ by their L2 configurations only:

1. Private L2 TLB: Each core has its own dedicated 128-entry, 8-way set associative L2 TLB. Both the core's L1 instruction and data TLBs are connected to this L2 TLB.

2. Shared L2 TLB, without a global ASID space: There is a unified 1024-entry, 8-way set associative TLB. Each TLB entry is tagged with an associated hart ID, in addition to an ASID and a virtual page number (VPN).

3. Shared L2 TLB, with a global ASID space: There is a unified 1024-entry, 8-way set associative TLB. Each TLB entry is tagged with only an ASID and a VPN. In this setup, the TLB is free to serve an entry that was originally created by one hart to a different hart (this is not possible in the previous configuration above).

While all setups have the same number of L2 TLB entries per core, their hardware costs differ: setup (2) requires additional tag bits for hart IDs; (1) has low requirement for communications; (2) and (3) need less logic for tag checking (as they are 8-way in total instead of 8-ways per core).

Figure 3.3 shows the results generated using the PARSEC benchmark suite to test the scenario where memory sharing is common and frequent. A shared L2 TLB without global ASIDs performs similarly or better compared to a private L2 TLBs in all PARSEC benchmarks. The average L2 miss rate drops from 29% for the private L2 TLB setup to 23%

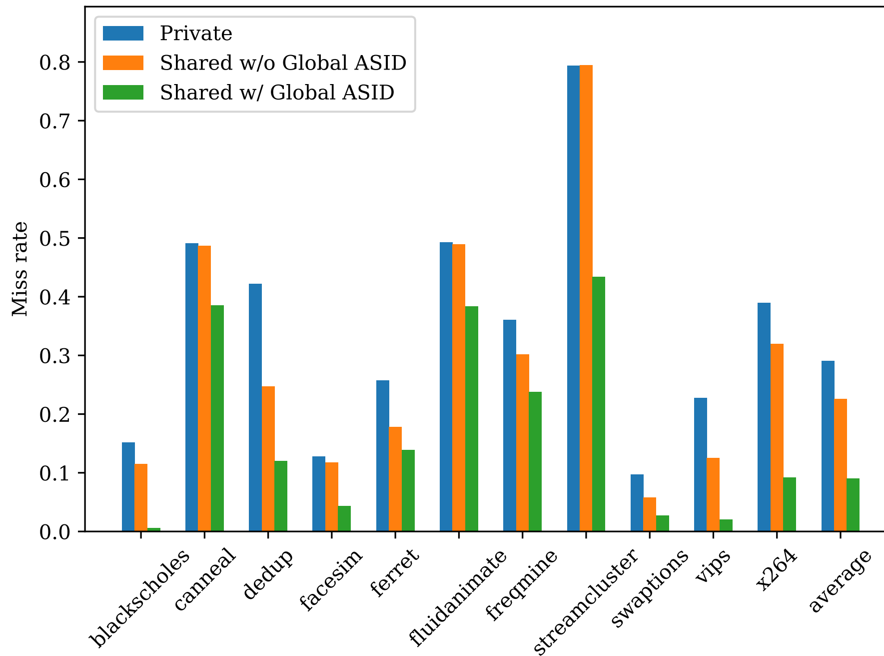


Figure 3.3 Local L2 TLB miss rates running the PARSEC benchmarks

for the shared L2 TLB setup without global ASIDs. A major improvement is observed in all benchmarks when the shared L2 TLB is allowed to exploit a global ASID space. The miss rate drops further to 9%, in line with my expectation. When sharing is common, a global ASID space can avoid storing redundant copies, therefore can cache more entries.

I also use Linux kernel compilation as a benchmark. Compiling a single C file is serial, but the whole task is embarrassingly parallel as different files can be concurrently compiled using `make -j<n>`. This benchmark tests the different TLB organisations when memory sharing and communications between cores are infrequent. Figure 3.4 shows the statistics gathered. When all 8 cores are used, the miss rate reduction is minimal as expected. When fewer jobs are used, there are less active harts but the shared TLB size remains the same – the dynamic partitioning nature of a shared TLB means that the effective per-hart L2 TLB size goes up, so we see a huge performance gain over the private L2 TLB. I believe that the shared TLB with global ASIDs outperforms the shared TLB without global ASIDs slightly as processes are occasionally migrated between harts. This result shows that shared TLBs can be particularly useful for low or moderate utilisation.

The relationship between miss rate and per-core L2 TLB size in these three setups is also explored, with the result shown in Figure 3.5. As I expected, the miss rate is negatively correlated to the per-core L2 TLB size in all three setups. A more interesting observation is that the miss rate of a shared TLB with global ASID is even better than the miss rate of

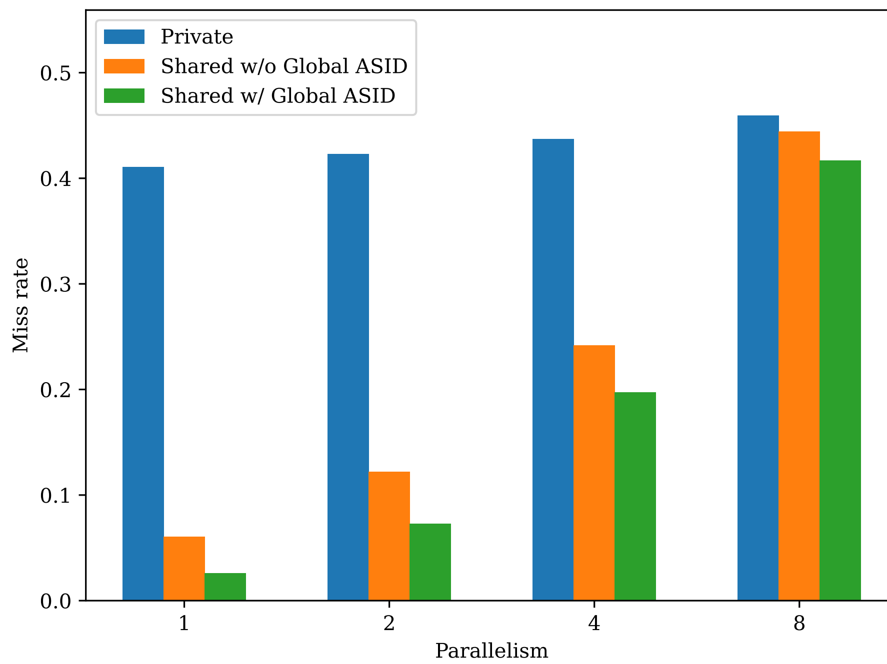


Figure 3.4 Local L2 TLB miss rates compiling the Linux kernel, in relation to parallelism

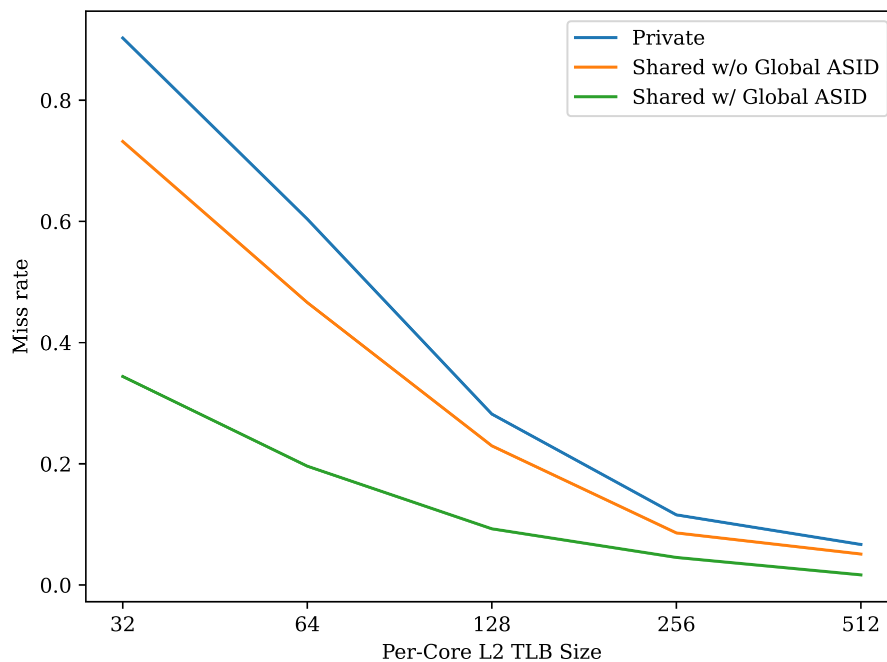


Figure 3.5 Local L2 TLB miss rates running the PARSEC benchmarks, in relation to per-core L2 TLB size

a shared TLB without global ASID with twice as many entries, and is approximately the same as the miss rate of private L2 TLB design with 4 times as many entries per core. This suggests that, in parallel workloads, if we change a private L2 TLB design to a shared L2 TLB design with global ASID, we will be able to save $3/4$ space while preserving the miss rate.

3.3.2 Global ASID

The empirical data from the last section shows a clear advantage of shared TLBs with global ASID space, and I believe that such design should definitely be allowed by the RISC-V specification. Global ASID space also has other potential benefits than just allowing shared translation caches. It also permits virtual cache, which is used in some GPU designs [123]. For OSes, the gain of independent hart-local ASID space has little value, and may not be worth the complexity of tracking ASID individually per hart. If the hardware supports remote TLB shootdown, by using a consistent meaning of ASIDs across harts, OSes can take the advantage transparently by issuing `sbi_remote_sfence_vma_asid` SBI calls and let the machine-level firmware choose the appropriate platform-specific mechanism to perform remote TLB shootdown.

To permit these designs, the RISC-V privileged specification requires a change. However, we cannot simply declare that now ASIDs are global – this will break existing software that uses ASID 0 on all harts, and we will lose the ability to perform a simple division of hardware resources with logical partitioning (LPAR) [22, 107] without hypervisor extension support. Following my suggestions about a global ASID space, the RISC-V's privileged specification committee has decided to place a forward-compatibility requirement on supervisor software so that non-zero ASIDs should have consistent meanings across different harts, and software should not use ASID 0 if it decides to use other ASIDs. This is backward-compatible with existing software; for hardware that does not exploit a global ASID space, machine-level code can still perform LPAR-style separation based on its knowledge of the hardware.

3.3.3 ASID Sharing Extension

In this section I describe a possible extension to the RISC-V privileged specification that would permit configurable ASID sharing [52], rather than having to choose global or hart-local ASIDs at design time.

A new CSR, MASI (Machine Address Space Isolation CSR) could be introduced, which is a write-any-read-legal (WARL) CSR register. This CSR can be implemented by all hardware implementations regardless of whether they would like to use global ASID space.

The number of writable bits is implementation-defined. Non-writable bits are hardwired to constants, which may or may not be zero. Less significant bits are implemented first. Software can determine the writable part of this CSR by first writing all zeroes to MASI and read back, then writing all 1s to MASI and read back, and XORing the results. Following this procedure, the bits containing 1 in the XORed result are writable.

<table>
<tr><td colspan="3">MASI mismatch</td><td>Cannot share; different ASID isolation domain</td></tr>
<tr><td rowspan="6">MASI match</td><td colspan="2">VMID mismatch</td><td>Cannot share; different VM</td></tr>
<tr><td rowspan="5">VMID match</td><td>The entry has global bit set</td><td>Can share</td></tr>
<tr><td>Both ASIDs are non-zero and identical</td><td>Can share</td></tr>
<tr><td>Both ASIDs are non-zero but not identical</td><td>Cannot share</td></tr>
<tr><td>Both ASIDs are zero</td><td>Cannot share; software does not support ASID</td></tr>
<tr><td>One ASID is zero and another is non-zero</td><td>Implementation-defined</td></tr>
</table>

Table 3.2 Look-up table for whether translation caches could be shared

Table 3.2 shows whether two entries could be shared under the proposed extension. I deliberately include the implementation-defined clause when one hart uses an ASID of zero and another hart uses a non-zero ASID while sharing the same MASIs and VMIDs. This relaxation allows simpler hardware implementations when handling ASID 0, allowing them to simply map ASID 0 to a unique global ASID (e.g. hart ID) instead of implementing complex logic to deal with ASID 0 specially in the entire TLB hierarchy.

This design opens up space for possible designs:

- If an implementation utilises global ASID space, but does not want to support LPAR, e.g. not needed or there is a hypervisor extension for the task, the implementation can simply hardwire MASI to 0.
- If an implementation does not exploit global ASID space, it can hardwire MASI to the hart ID (or any unique number).
- If an implementation wants to utilise global ASID while supporting LPAR, it will need to implement writable bits in this CSR. It also needs to add corresponding tag bits in TLB entries. Note that this is not an additional overhead as without this extension and global ASID, it would need to tag TLB entries with hart IDs anyway.

The proposed extension also allows other topologies. For example, if there are two sockets and ASID sharing is supported within a socket, then the hardwired constant bits can be set to different values on two sockets, so the software would know that some harts can share a translation cache but not others.

The proposed extension requires software to use non-zero ASIDs and global pages to represent consistent meaning across harts, unless they are aware of the MASI configuration;

this is consistent with the aforementioned forward-compatibility requirement in place. The proposed change will also not affect existing hardware, as accesses to the proposed CSR will trigger illegal instruction faults. The firmware could catch the exception and return the hart ID instead. Moreover, I expect that most implementations will not need to support both ASID sharing and LPAR, so they would only need to implement a read-only CSR, which is almost free in hardware.

As I have no need for LPAR, in all the work that follows I simply utilise a global ASID space without implementing any MASI bits.

3.4 Classifying Synonyms

The TLBSim tool was also used to investigate how Linux currently makes use of synonyms. In order to detect synonyms, the ideal TLB was modified to also keep a reverse translation, stored in a multi-map. When a new translation entry is added to the TLB, the reverse translation is looked up. If there is already a reverse translation, a synonym is considered to be found.

By analysing the traces, I observe that there are three common use cases of synonyms:

- Shared libraries, such as `ld.so` and `libc.so` have only one copy in memory, and they are mapped into the address space of different processes (thus different ASIDs). When address space layout randomization (ASLR) is off, many of them are mapped to the same virtual addresses for different processes of the same executable. The shared libraries may be mapped to different virtual addresses for different executables. If ASLR is on, their virtual address mappings are randomised and will therefore most likely be different for multiple instances of the same executable.

- Forking also results in synonyms. When `fork` is called, Linux will mark all non-shared pages as read-only and duplicate the memory context (and thus allocate a fresh ASID). This introduces synonyms as the backing physical pages are not actually duplicated.

- Memory mapped files created by `mmap`. If a single file is mapped multiple times, the same physical page is used for all the mappings, creating a synonym. This is read-only for a non-`PROT_WRITE` mapping. For `PROT_WRITE` mappings, if `MAP_PRIVATE` is specified, the copy-on-write mechanisms will still cause the page to be mapped as read-only.

It can be observed that those commonly observed synonym scenarios almost always create read-only synonyms. The virtual address given cannot be used to write to the backing physical memory directly.

The following scenarios could possibly lead to read-write synonyms being introduced:

- Memory mapped file created by a `PROT_WRITE` *and* `MAP_SHARED` `mmap`. If the mapping is shared between processes via independent opening of the same backing file, then the virtual address may be different. If the mapping is shared by forking, then the virtual address will be identical.

- Explicit shared memory via `shm_open` routines. In practice, GLIBC implements this function by opening/creating a file located in `/dev/shm`, which is just a mounted in-memory temporary file system. Therefore from the kernel's perspective, this is no different from the bullet point above.

I observe that shared read-write mappings require explicit creation. Most usages of `MAP_SHARED` and `PROT_WRITE` memory maps are used for fast file writes instead of communicating between two processes. Some applications, such as Wayland, does make use of shared memory for buffers to avoid transferring heavy amount of data over sockets. In Wayland's usage scenario, double buffering is used and the buffer switching is communicated over sockets, so the shared memory is not used for synchronisation and at a specific time, only one process will access a particular memory region.

Overall the observation shows that: most synonyms are read-only, while a limited number of applications will make use read-write synonyms. It is also possible to have one address writeable and another read-only, but I believe the case is rare, and even though we try to perform correctly even in this case, performance should not be a concern in this case.

3.5 R2VM – Fast Cycle-Level Instruction Simulator

TLBSim was initially explored as the basis for simulating cache coherence protocols. However, due to the inherent limitation of TLBSim being a TLB simulator, I can only use 4KiB cache lines, which is very different from the cache line that is commonly used (64B). Moreover, TLBSim is not a timing simulator and multiple hardware threads run on different threads, so the inter-core effect of cache coherence protocol cannot be observed.

I experimented with simulators such as gem5 [18], but found its performance a major limiting factor, given the need to consider benchmarks that involved the operating system. For example, the smaller synthetic microcontroller benchmark CoreMark [32] executes at $\sim 10^8$ instructions per iteration, while SPEC2017 [34], a larger and more realistic benchmark running real-life applications, requires an order of 10^{12} instructions for a single run [78]. SPEC takes from hours to days to run even on real machines, and hardly possible for simulators to run to completion. PARSEC's `simlarge` dataset is a little bit smaller than

SPEC2017, requiring a magnitude of 10^{11} instructions for a single run. It is therefore helpful to have a fast simulator.

I evaluated `riscv-isa-sim`, or *spike*, which is faster than gem5, but I found that it has no support for a processor pipeline model or cache coherency, and its scheduling of hardware threads is coarse-grained (every 1000 instructions in the default configuration). QEMU is even faster but as discussed earlier it is also a poor fit for timing simulation purposes.

Of course, fast simulation involves a trade-off between the fidelity of the model and the speed at which simulations can be completed. Unfortunately, we are currently forced to choose between slow cycle-accurate simulators or fast functional-only simulators such as QEMU. In particular, there is a lack of fast full-system simulators that can accurately model cache-coherent multi-core processors.

I therefore developed Rust RISC-V Virtual Machine (R2VM), a fast cycle-level multi-core simulator for RISC-V systems. R2VM is written in the increasingly popular high-level system programming language Rust [114]. I selected Rust for this project because the Rust language guarantees data-race-freedom in compile time. R2VM is released under permissive MIT/Apache-2.0 licenses on GitHub [49]. R2VM uses binary translation to speed up execution; and to my knowledge, this is the first binary translated simulator that supports cycle-level multi-core simulation. It can accurately model cache coherence protocols and shared caches. Cycle-level simulations are possible at more than 20 MIPS, while the performance of functional-only simulations can outperform QEMU and exceed 400 MIPS.

3.5.1 Implementation Overview

A high-level overview of R2VM is shown in Figure 3.6. R2VM uses binary translation to speed up execution. The high-level control flow of R2VM, as shown in Figure 3.7, is similar to other binary translators. When an instruction at a particular PC is to be executed, the code cache is looked up and the cached translated binary is executed directly if found; otherwise, the binary translator is invoked and an entire basic block is fetched, decoded, and translated.

The binary translation component builds upon my own earlier work, `riscv-dbt` [47], a userspace binary translator, although I reimplemented it in Rust rather than using the original C++ codebase. `riscv-dbt` contains a template-based binary translator and a graph-IR-based binary translator, and the latter incorporates various optimisation and transformation passes. However, I have only adopted template-based binary translation for R2VM because existing `riscv-dbt` optimisations rely on assumptions that work well with userspace programs but not with system-level binary translators due to the existence of paging, traps and interrupts.

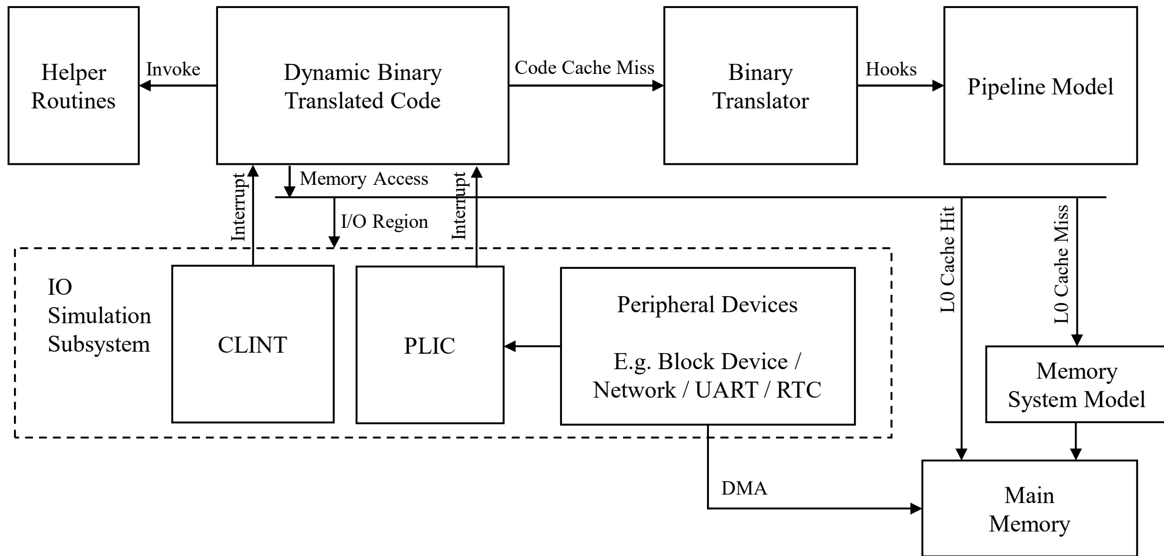


Figure 3.6 High-level overview of R2VM components

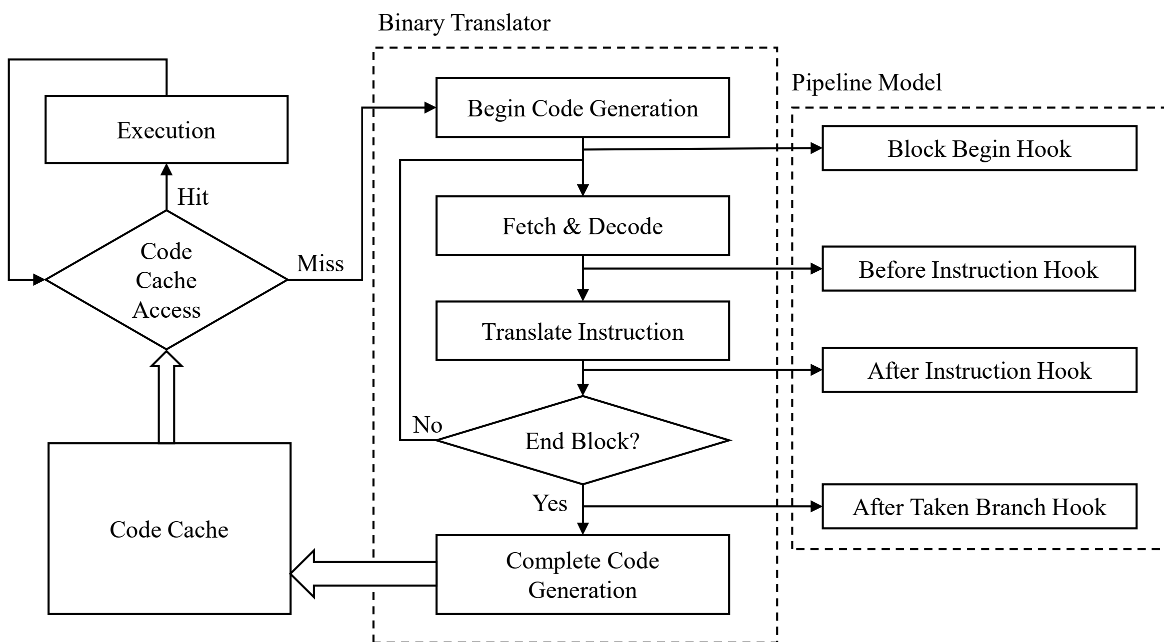


Figure 3.7 Control flow overview of R2VM

R2VM is capable of performing user-level simulation, supervisor-level simulation and machine-level simulation. For user-level simulation, Linux system calls (syscalls) are emulated, and for supervisor-level, supervisor binary interface (SBI) calls are emulated.

3.5.2 Pipeline Simulation

One key difference between R2VM and existing emulators that exploit binary translation, such as QEMU, is that R2VM allows for the construction of a pipeline performance model. This is achieved by providing a number of opportunities where hooks provided by the pipeline model can be invoked, as shown in Figure 3.7. Hooks can process relevant instructions and generate essential microarchitectural simulation code if necessary. The hooks can also indicate the number of cycles it would take for the instruction to complete. It should be noted that this is only for the execution pipeline; the memory system, including caches, is modelled in a separate component.

For simple models, such as gem5's "timing simple" model where each instruction takes 1 cycle to execute, implementation is straightforward as shown in Listing 3.2.

```
1  #[derive(Default)]
2  pub struct SimpleModel;
3
4  impl PipelineModel for SimpleModel {
5      fn after_instruction(&mut self, compiler: &mut DbtCompiler, _op:
   ↪ &Op, _compressed: bool) {
6          compiler.insert_cycle_count(1);
7      }
8
9      fn after_taken_branch(&mut self, compiler: &mut DbtCompiler, _op:
   ↪ &Op, _compressed: bool) {
10         compiler.insert_cycle_count(1);
11     }
12 }
```

Listing 3.2 Timing simple model implementation

I have also implemented and validated an in-order pipeline model that accurately models a classic 5-stage pipeline with a static branch predictor. My implementation captures pipeline hazards, such as data hazards caused by load-use dependency and stalls due to a branch/jump into a misaligned 4-byte instruction. Unlike Böhm et al.'s simulator [20] which needs to call a "pipeline" function after each instruction, my implementation models pipeline behaviours

during DBT code generation and reflects them as number of cycles taken, therefore requires no explicit code to be executed at run time.

It is possible to simulate more complex pipeline models, either by making an estimation of pipeline states (and sacrifice some accuracy) or generating custom assembly in the hooks to maintain these states during execution (and sacrifice some performance). For the purpose of investigation carried out in this thesis, I believe that an in-order 5-stage pipeline model is sufficient and did not implement more complex models.

3.5.3 Multi-core Simulation

The techniques described in the previous section work well for single-core systems. The binary translation implementation in R2VM is thread-safe, meaning that I could simply spawn multiple threads and run each core in its own thread. As described earlier, simply running them in parallel or synchronising them in a coarse-grained manner would sacrifice the fidelity of the simulation of a multi-core system. The ideal scheduling granularity is a cycle, i.e. having all simulated cores run in lockstep. This is, however, difficult to achieve for binary translators.

I experimented with the idea of using thread barriers to synchronise multiple threads, each simulating a single core. However inter-core communication via shared memory is not cheap, especially given the desired granularity, where there is very little work to perform between two barriers. Experiment shows that on a quad-core 3.4GHz Intel i7-6700 CPU, only $\sim$1 million synchronisations are possible per second, even after careful optimisation at the assembly level.

Lockstep Simulation

The approach I use takes inspiration from fibers, sometimes also referred to as coroutines or green threads. Fibers are cooperatively scheduled by the user-space application, and they voluntarily "yield" to other fibers, in contrast to traditional threads which are preemptively scheduled by the operating system and are generally heavy-weight constructs. Fibers are often used in I/O heavy, highly concurrent workloads, such as network programming, but this time I borrowed the idea for simulation.

In my implementation, I create one fiber for each hardware thread simulated, plus a fiber for the event loop. Each time the pipeline model instructs the DBT to wait for a few cycles, the matching number of yield calls will be generated. Listing 3.3 shows an example of generated code under timing simple model.

```
1  mov     rax, qword [rbp+0x78]  ; \
2  mov     qword [rbp+0x70], rax  ; | add a4, zero, a5
3  call    fiber_yield_raw        ; /
4  mov     eax, dword [rbp+0x78]  ; \
5  add     eax, -0x1              ; |
6  cdqe                           ; | addiw a5, a5, -1
7  mov     qword [rbp+0x78], rax  ; |
8  call    fiber_yield_raw        ; /
9  mov     eax, dword [rbp+0x70]  ; \
10 imul    eax, dword [rbp+0x50]  ; |
11 cdqe                           ; | mulw a0, a4, a0
12 mov     qword [rbp+0x50], rax  ; |
13 call    fiber_yield_raw        ; /
```

Listing 3.3 Example of generated code with yield calls. RBP points to architectural states, including an array for registers.

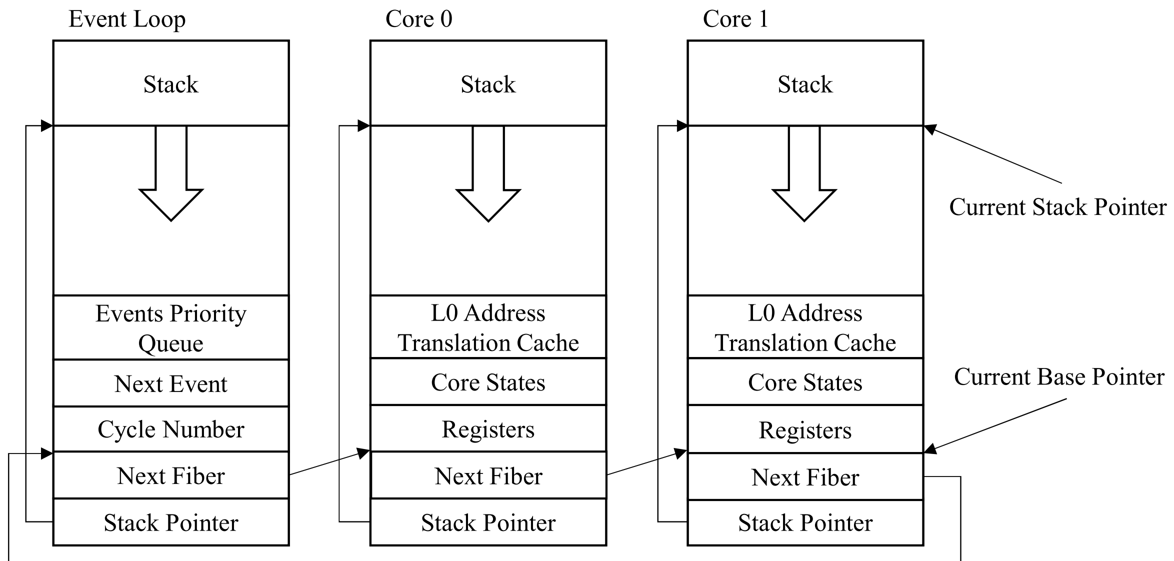


Figure 3.8 Memory layout of fibers

Instead of using a general fiber implementation, fibers in R2VM uses a specially engineered memory layout shown in Figure 3.8. Each fiber is allocated with a 2M memory aligned to 2M boundary, and the stack for running code on the fiber is contained within this memory range. The alignment requirement allows the fiber's start address to be recovered from the stack pointer by simply masking out the least significant 21 bits. The base pointer points to the end of fixed fiber structures rather than the beginning, so that positive offsets from the base pointer can be used freely by the DBT-ed code, while the negative offsets are used for fiber management.

The ABI of the host platform for DBT-ed code is not respected; I rather specify all registers other than the base pointer and stack pointer to be volatile, or caller-saved. By doing so, `fiber_yield_raw` does not need to bear the cost of saving any registers. To yield in non-DBT-ed code, all ABI-specified callee-saved registers are pushed into the stack before switching and restored afterwards.

This careful design makes fiber switching lightning fast; the `fiber_yield_raw` function is as simple as 4 instructions on AMD64, shown in Listing 3.4.

```
1 fiber_yield_raw:
2     mov [rbp - 32], rsp ; Save current stack pointer
3     mov rbp, [rbp - 16] ; Move to next fiber
4     mov rsp, [rbp - 32] ; Restore stack pointer
5     ret
```

Listing 3.4 Implementation of the fiber yielding code

Synchronisation Points

Simply yielding a few cycles after every executed instruction will severely limit performance and in many cases will be unnecessary. In practice, we only need to synchronise at points where the execution pipeline can produce visible side-effects to other cores and/or the rest of the system, or where the rest of the system's behaviour would affect the running pipeline.

I observed that there are three ways that a pipeline interacts with another:

- A memory operation is performed.
- A control register operation is performed. This includes read/write to performance monitor registers, or control registers related to the memory system.
- An interrupt happens.

For the first two types of interaction, a synchronisation point is inserted before and after memory operations are executed. For the third case (interrupts), because it is generally difficult to interrupt an DBT-ed code mid-way, I choose to check for interrupts only at the end of basic blocks. I believe that this decision will not affect the accuracy of the simulation due to the inherent entropy of I/O operations.

The positioning of yielding that lies in between two synchronisation points, therefore, would have no visible side-effects and cannot be distinguished. My implementation postpones all yielding until the next synchronisation point. The implementation shown in Listing 3.4 was modified slightly to allow multi-cycle yielding, and it demonstrates around 10% performance gain compared to naive yielding.

3.5.4 Memory Simulation

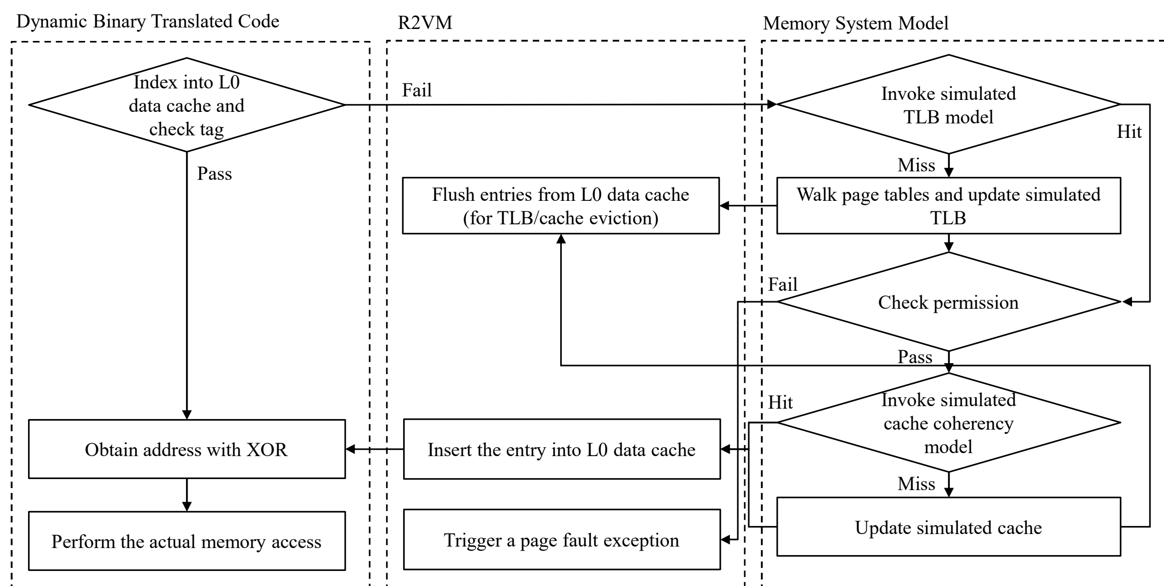


Figure 3.9 Control flow for memory accesses

Previous sections described how each core's pipeline is simulated and modelled, and how lockstep simulation is achieved. The techniques described and implemented above can speed up pipeline simulation significantly; but when the memory system is also simulated, the overall speedup can be very limited if all memory accesses need to go through the memory system model and be simulated. Moreover, instruction caches and their associated TLBs would also need to be simulated for accuracy, leading to even more overhead if the memory system model is to be invoked for each simulated instruction fetch.

L0 Data Cache

For memory operations, each running core has its own "L0 data cache". This L0 data cache is a structure in the simulator that can be accessed directly from the DBT-ed code and not part of the memory system model. When a core needs to read from or write to a memory address, it first checks if it is in the L0 data cache. If it hits, then memory access is performed entirely within DBT-ed code, bypassing the memory system model entirely.

As a result, the memory system model will not intercept all memory accesses. It is therefore important to control what could be in the L0 data cache. I maintain a property that if an access hits the L0 data cache, then it must be a cache hit in both the L1 data TLB and the L1 data cache, would the memory access reach the memory system model. The concept is similar to the way that I speed up TLB simulation as described in Section 3.1.1.

In the previous TLB simulation work, the property mandates an invariant that all entries in the L0
cache ent
shown in
model, co
property.

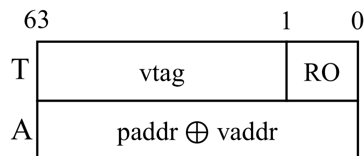


Figure 3.10 Memory layout of a tag entry in the L0 data cache

I carefully engineered the memory layout of L0 data cache entries for maximum efficiency. The L0 data cache is direct-mapped, with each entry representing a cache line. Each entry has a memory layout like Figure 3.10, comprising two machine words. It does not store actual memory contents; it rather stores a translation from the virtual tag to a physical Address. In a sense, it is more like a TLB with cache-line granularity than a cache. I pack the guest virtual address tag (`vtag`) together with a single bit (`RO`) indicating if the cache line is read-only into the first machine word `T`. For the second machine word `A`, instead of storing the guest physical address, the XOR-ed value of the guest physical address (`paddr`) and the corresponding guest virtual address (`vaddr`) is stored instead.

For each memory access, the L0 data cache is indexed into using the virtual tag. An example of the assembly code generated for a read access is shown in Listing 3.5. For read accesses, as the `RO` bit does not matter, the tag check is performed by comparing `T >> 1` with the vtag, ignoring the `RO` bit. If the check passes, the requested virtual address is XOR-ed

```
1       ...
2       ; Virtual address in RSI
3       mov     rcx, rsi                ; \
4       shr     rcx, GRANULARITY        ; / Compute virtual tag
5       mov     eax, ecx                ; \
6       and     eax, 1023               ; | Compute array offset
7       shl     eax, 4                  ; /
8       mov     rdx, [rbp+rax+0x140]    ; \
9       shr     rdx, 1                  ; |
10      cmp     rdx, rcx                ; | Check if tag matches
11      jne     .Lslow                  ; /
12      xor     rsi, [rbp+rax+0x148]    ;   Compute physical address
13  .Lfin:
14      ; Physical address in RSI
15      ...
16  .Lslow:
17      mov     rdi, rbp                ; \
18      xor     edx, edx                ; | Refill the translation cache
19      call    translate_cache_miss    ; /
20      mov     rsi, rdx                ; \
21      test    al, al                  ; | Resume if no trap happens
22      je      .Lfin                   ; /
23      ; Update performance monitor
24      jmp     helper_trap             ;   Trap for page or access fault
```

Listing 3.5 Generated assembly sequence for memory load

with `A` to produce an address that can be accessed directly within DBT-ed code. The XOR operation changes the page number from virtual to physical while keeping the page offset intact. If the check fails, the control flow jumps to the cold path and the memory system model is invoked. The memory system model will simulate both the TLB and the data cache, and will either trigger a page fault, or insert an entry into the L0 data cache and resume execution of DBT-ed code. For write access, we can check that `RO` bit is zero by simply comparing if `vtag << 1` is equal to `T`. An example of the generated assembly instruction sequence is shown in Listing 3.6.

```
1      ...
2      ; Virtual address in RSI
3      mov     rcx, rsi              ; \
4      shr     rcx, GRANULARITY      ; / Compute virtual tag
5      mov     eax, ecx              ; \
6      and     eax, 1023             ; | Compute array offset
7      shl     eax, 4                ; /
8      add     rcx, rcx              ; \
9      cmp     rcx, [rbp+rax+0x140]  ; | Check if tag matches
10     jne     .Lslow                ; /
11     xor     rsi, [rbp+rax+0x148]  ;   Compute physical address
12 .Lfin:
13     ; Physical address in RSI
14     ...
15 .Lslow:
16     mov     rdi, rbp              ; \
17     mov     edx, 1                ; | Refill the translation cache
18     call    translate_cache_miss  ; /
19     mov     rsi, rdx              ; \
20     test    al, al                ; | Resume if no trap happens
21     je      .Lfin                 ; /
22     ; Update performance monitor
23     jmp     helper_trap           ;   Trap for page or access fault
```

Listing 3.6 Generated assembly sequence for memory store

The existence of the L0 data cache means that for the fast path, a simulated memory operation can be completed with just 3 real memory accesses. In the default configuration, because the memory system model does not intercept all memory accesses, replacement policies such as LRU cannot be used for the simulated TLB and cache. Generally, I believe this is an acceptable accuracy loss to trade for vastly better simulation performance. If

LRU-like policies are really needed, the L0 data cache could be bypassed and the memory system model be invoked for each memory access, in sacrifice of performance.

L0 Instruction Cache

R2VM also simulates instruction TLBs and caches in a similar fashion to the data cache. Each core also has its own L0 instruction cache, with a simpler entry layout because read/write permission needs not to be distinguished. To keep the overhead of simulating the instruction cache down, instead of accessing it each time an instruction is executed, it is instead only accessed when a basic block begins, or when the instruction being translated is in a different cache line compared to the previous instruction. For a cache line size of 64 bytes, this means that I only need to generate a single L0 instruction cache access for every 16-32 instructions.

The L0 instruction cache is also used for optimisation of cross-page instructions. This is described later at Section 3.5.6.

I also creatively use the L0 instruction cache to optimise jumps across pages. Traditionally, because the page mapping might change and therefore the actual target of jump instruction might change, DBTs have to conservatively not link these blocks together. Instead, we can check the L0 instruction cache (which we would need to check anyway when next block begins) and see if the target is the same as the cached target. If so, the cached target is used and the control does not go back to the main loop.

Cache Coherency

The memory system design described above inherently supports the use of cache coherency. Whenever the cache coherence protocol invalidates a cache line in the L1, the same cache line can be easily removed from the L0 data cache of the relevant core by setting the corresponding `vtag` to an invalid value. Because all simulated cores execute in lockstep, and there are synchronisation points before all memory accesses, the effect of the invalidation will be visible before the next memory access.

3.5.5 Runtime Reconfiguration

In many cases, we want to gather cache statistics with the behaviour of the OS taken into account, but we do not want to count the OS booting and workload preparation steps before the region of interest, and do not want to pay for the performance overhead of detailed models for these portions. The design of R2VM takes this into account, and both pipeline and memory system models can be switched dynamically in the runtime. The switching is controlled by writing a special CSR in the vendor-specific CSR range.

R2VM supports pipeline model switching by simply flushing the code cache for the translated binary, and then ensuring that the DBT engine uses the new model's hooks for further code generation. As each core has its own DBT code cache, pipeline models can be specified for each core rather than requiring all cores to use the same model.

Memory system model can also be switched dynamically. When switching is requested, all harts are stopped. The code cache and all L0 data and instruction caches are flushed. All harts are then restarted with the new memory system model installed. The granularity of memory system model simulation is also a runtime-configurable property; this property determines the cache line size of the L0 data cache. For example, if both the TLB and the cache are simulated, the granularity can be set to 64 bytes. If only the TLB is simulated, the granularity can be set to 4096 bytes, turning the simulator's L0 data cache effectively into an L0 data TLB.

R2VM can also switch between lockstep execution and parallel execution like other binary translators during the runtime if the memory system model permits. Parallel execution, when paired with the "functional" pipeline model, behaves functionally equivalent to QEMU and gem5's "AtomicSimpleCPU"' model which permits fast-forwarding of the aforementioned booting and preparation steps.

3.5.6 Other Techniques and Design Decisions

Code Cache and Paging

Paging is the biggest difference between a user-space binary translator and a full-system binary translator. In user-space binary translators, such as `riscv-dbt`, code caches are indexed by virtual addresses. Page mappings are handled by the operating system and is thus transparent to the user-space programs in execution. For full-system binary translation however both physical and virtual addresses are relevant, and therefore if the code cache is indexed by virtual addresses, any change of page mappings could potentially void the cached translated code. To prevent the need to flush the entire translation cache when the page table root pointer is changed (via writing to SATP), the code cache is indexed by physical addresses.

This is however still insufficient to guard against self-modifying code. RISC-V does require the use of SFENCE.VMA (for TLB flushing) or FENCE.I (for instruction cache flushing) instructions after the code modifications to make changes visible, but flushing the entire code cache for these instructions is still undesirable, given that SFENCE.VMA instruction can be executed quite commonly, e.g. stacks or allocated memory can be overcommitted

and only mapped lazily when they are actually used. R2VM is designed to avoid code cache flushes in these circumstances as well.

This is achieved by maintaining a B-tree map containing all pages that currently have a cached translation in the code cache. No pages in this B-tree map are allowed to be modified. Since the fast path for memory access is entirely within the binary translated code and bypasses any interpreter or helper function, this requirement implies that no pages in this B-tree map can have any sub cache lines present in the L0 data translation cache with write permission. Whenever the slow path of memory store hits (`translate_cache_miss` is called in Listing 3.6), the address is checked against these "protected" pages. If the target address is indeed protected then it will trigger a code cache invalidation. All harts will be paused after execution of the current basic block completes and have affected cached code invalidated by patching the first instruction to a `RET` instruction. Whenever a new page is being "protected", the L0 data translation caches of all harts are scanned and any conflicting cache lines are removed. "Stopping the world" is avoided for this step by serializing access to the B-tree map with a mutex.

Code Cache Size Management and Sharing

The management of the size of the code cache is done in a similar fashion to that of QEMU. Each hart has a fixed-size code cache pool, and memory blocks are allocated with bump allocation. Whenever the code cache memory pool is exhausted, the code cache is flushed in its entirety. Although probably not the most efficient way to manage the code cache, it greatly reduces the complexity especially when block chaining is implemented and a translated code block can reference another.

Cota et al. [35] suggests sharing a code cache between multiple cores to promote code reuse and boost performance. In contrast, I have decided to provide each hardware thread its own code cache. This allows different code to be generated for each core, e.g. in the case of heterogeneous cores. This also makes code self-modification possible (e.g. block chaining), as synchronising code modification when another thread is potentially running the code is problematic.

Speculative Block Chaining

Block chaining is an old performance improvement technique that has existed in QEMU [13] and other binary translators for a long time. Without block chaining, a branch or jump instruction is translated to code that updates the PC and then returns the control flow to an outer loop which looks up the new PC from the code cache, performs binary translation if

misses, and then jumps to the translated code. Block chaining removes the extra lookup if the jump or branch target is a fixed address; instead of returning, a trampoline is generated initially; lookup is performed once, and the trampoline is overwritten with a jump instruction to the looked-up address. So from the second time onward, the code cache lookup is bypassed.

```
a_translated:
  ...
  ; Generated assembly for `j b`
  ; Adjust PC
  add  qword [rbp+0x100], b - a
  ; Unpatched; Helper call
  call helper_patch_direct_jump

; b may yet to be translated
```

```
a_translated:
  ...
  ; Generated assembly for `j b`
  ; Adjust PC
  add  qword [rbp+0x100], b - a
  ; Patched; direct jump
  jmp  b_translated

b_translated:
  ...
```

Listing 3.7 Example assembly changes with block chaining, before and after patching

R2VM implements block chaining as shown in Listing 3.7. However as mentioned in Section 3.5.6, the code cache is indexed by physical addresses, and the block chaining can only be used if the target address resides in the same page as the current address. For cross-page jumps though, despite the technical possibility of a change in the physical address of the target, in practice it is highly unlikely to change. R2VM exploits this knowledge by speculatively performing the block chaining for cross-page jumps, verifying the correctness and bail out if the jump turns out to be incorrect.

Listing 3.8 shows an example of generated code with speculative block chaining. A special preamble exists before the actual code of a basic block and serves as the speculative entry point. It performs a lookup into the L0 instruction translation cache, similar to the listings described in Section 3.5.4. The translated physical address is then compared to the known actual physical address of this basic block. In case of a mismatch, a helper is called which looks up the correct target and re-patch the speculative jump instruction (address of which is stored in ‘rbx‘, a non-volatile register).

The technique is also used for indirect jumps. In many indirect jump cases the target is still fixed (e.g. `AUIPC + JALR` sequence, dynamic linking), or rarely change (e.g. devirtualising a virtual function call), and therefore providing significant speedup.

Cross-page Instructions

Apart from cross-page jumps, a trickier situation is dealing with a 4-byte uncompressed instruction spanning two pages. This is not an issue for user-space binary translators, but

```
1  a_translated:
2      ...
3      ; Generated assembly for `j b`
4      ; Adjust PC
5      add     qword [rbp+0x100], b - a
6      ; Address for patching, still needed after patching for
   ↪  misprediction fixup
7      lea     rbx, [.Lafter_call]
8      ; Patched; speculative direct jump
9      ; This is `jmp helper_pred_miss` before patching for a forced
   ↪  prediction miss
10     jmp     b_translated_speculative
11 .Lafter_call:
12
13 b_translated_speculative:
14     ; Translate PC to physical address with the L0 instruction cache
15     mov     rsi, qword [rbp+0x100]
16     mov     rcx, rsi
17     shr     rcx, GRANULARITY
18     mov     eax, ecx
19     and     eax, 1023
20     shl     eax, 4
21     cmp     rcx, [rbp+rax+0x4140]
22     jne     .Lslow
23     xor     rsi, [rbp+rax+0x4148]
24 .Lfin:
25     ; Check if the physical address is the one originally used for
   ↪  binary translating this block
26     cmp     rsi, B_PHYSICAL_ADDR
27     jne     helper_pred_miss
28 b_translated:
29     ...
30 .Lslow:
31     mov     rdi, rbp
32     call    insn_translate_cache_miss
33     mov     rsi, rdx
34     test    al, al
35     je      .Lfin
36     jmp     helper_trap
```

Listing 3.8 Example patched assembly with speculative block chaining

as discussed in Section 3.5.6, R2VM is indexed by physical addresses, and we will have to guard against cases where the page mapping changes without any changes to physical memory occupied by the code. We could bail out to an interpreter each time such instructions are executed, but doing so would be very inefficient.

```
1      mov     rsi, qword [rbp+0x100] ; \ Compute the address of the
2      add     rsi, -2                ; /   second half
3      mov     rcx, rsi               ; \
4      shr     rcx, GRANULARITY       ; |
5      mov     eax, ecx               ; |
6      and     eax, 1023              ; | Translate the address to
7      shl     eax, 4                 ; |   physical address
8      cmp     rcx, [rbp+rax+0x4140]  ; |
9      jne     .Lcache_slow           ; |
10     xor     rsi, [rbp+rax+0x4148]  ; /
11 .Lcache_fin:
12     cmp     word [rsi], 0xCCCC ; \ Check the first half of the
13     jne     .Lmiss             ; /   instruction is expected
14 .Lreserve:
15     jmp     .Lmiss             ; \
16     nop                        ; | Reserve space for translating
17     ...                        ; |   the instruction
18     nop                        ; /
19 .Lmiss:
20     movzx   ecx, word [rsi]    ; \
21     shl     ecx, 16            ; | Assemble the actual instruction
22     or      ecx, LO_BITS       ; /
23     mov     rdi, rbp           ; \
24     lea     rdx, [.Lreserve]   ; | Recompile into the reserved space
25     call    icache_cross_miss  ; /
26     jmp     .Lreserve          ;   Jump to the translated instruction
27 .Lcache_slow:
28     mov     rdi, rbp                       ; \
29     call    insn_translate_cache_miss ; |
30     mov     rsi, rdx                       ; |
31     test    al, al                         ; | Slow path for I$ access
32     je      .Lcache_fin                    ; |
33     jmp     helper_trap                    ; /
```

Listing 3.9 Example of generated assembly for a cross-page instruction

Listing 3.9 shows how R2VM generates code for a cross-page instruction. Whenever such an instruction is to be executed, the actual second half (the 2 bytes that lies in another page) of the instruction is fetched, with the help of the L0 instruction cache. If the actual second half differs from the expected second half used when the instruction is compiled, or the stub is executed in the first time, a prediction miss happens and a helper is called to recompile the instruction into a reserved space that is large enough for all instructions.

In practice sometimes the subsequent page is paged-out and the trap path is hit; the miss path is hardly exercised after initial compilation. The technique can prevent the control flow from leaving binary translated code in most cases.

3.5.7 Evaluation

As described in Section 3.5.1, R2VM offers a range of pipeline models and memory system models to select from, and allows switching between them mid-simulation. Each model shows different trade-offs. A list of the pipeline and memory system models that were implemented is given in Table 3.3 and Table 3.4.

Name	Description
Functional	Cycle count not tracked
Simple	Each non-memory instruction takes one cycle
InOrder	Models a simple 5-stage in-order scalar pipeline

Table 3.3 List of pre-implemented pipeline models

Name	Description
Functional	Memory accesses not tracked
TLB	TLB hit rate collected; cache not simulated
Cache	Cache hit rate collected; TLB and cache coherency not modelled; parallel execution allowed
MESI	A directory-based MESI cache coherence protocol with a shared L2. Lock-step execution required.

Table 3.4 List of pre-implemented memory system models

Accuracy and Validation

For pipeline models, I validated the accuracy of the in-order model against an actual RTL implementation of a RISC-V core using CoreMark [32]. CoreMark is particularly helpful for

this validation, as CoreMark's working set is small enough to fit into caches and therefore the memory system of the RTL implementation would not affect the benchmark result. In a run, the RTL implementation reports 2.10 CoreMark/MHz where the in-order model, when paired with the functional memory system model, reports 2.09 CoreMark/MHz. The difference is less than 1%. The "simple" model is simply validated by checking that all cores have their MCYCLE and MINSTRET CSR equal.

For memory system models, I used a few micro-benchmarks to cover the use case for each model. For TLB and cache simulation, I used a single-core micro-benchmark that is similar to the MemLat tool from the 7-zip LZMA benchmark [93]. For the modified, exclusive, shared and invalid (MESI) cache-coherency model, I used a micro-benchmark to simulate a scenario where two cores are heavily contending over a shared spinlock. The memory system model under test is used together with the validated in-order pipeline model, and I compare the number of cycles taken to execute a benchmark in R2VM and in RTL simulation. The error is up to 3% for single-core non-coherent models or low-contention cases, and can be up to 10% when high-contention cases are simulated with cache coherency. I believe that the major source of inaccuracy comes from the fact that all transactions simulated are atomic: in R2VM, the interconnect is not modelled in detail; a message with a multi-cycle delay is simulated as an atomic message followed by a multi-cycle sleep. Although not as accurate as the pipeline model, I believe at this accuracy the simulation can provide sufficiently representative metrics with typical low-contention workloads for exploring design decisions.

Performance

I evaluated the performance of R2VM against QEMU using the `dedup` workload from PARSEC [17] on 4 cores to test simulator performance. `dedup` is selected because it is one of the few PARSEC benchmark that does not rely on floating-point instructions. Both R2VM and QEMU interpret floating-point operations in software instead of translating them due to floating point unit (FPU) incompatibility between different architectures. Selecting a benchmark that contains mostly integer instructions and memory operations avoids potential distortion caused by the difference in softfloat implementation. The kIPS numbers of the gem5 simulator are from Saidi et al.'s presentation [104].

As shown in Figure 3.11, the techniques I use lead to superb performance. When caches are not simulated and therefore cores can run in parallel threads, R2VM runs at $>$300 MIPS per core, even outperforming QEMU. Lockstep execution brings down performance by 10x to $\sim$30 MIPS (for 4 simulated cores in a single-threaded), but this is still significantly faster than gem5.

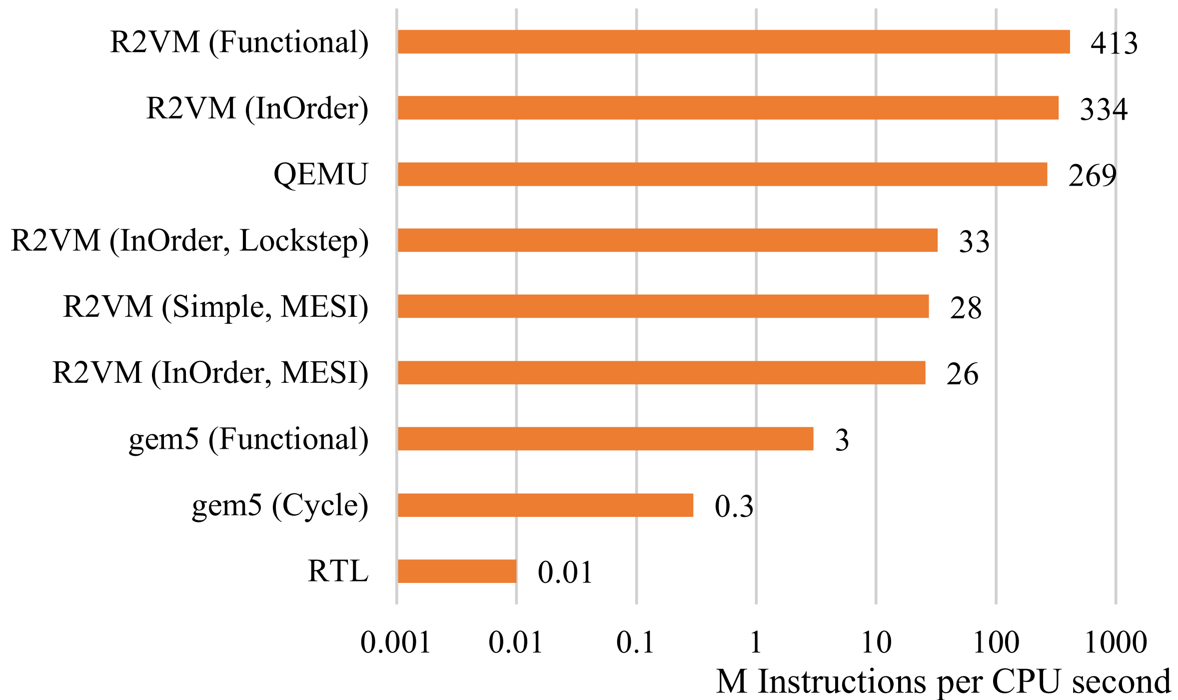


Figure 3.11 Performance comparison between models and other simulators. R2VM runs use functional models if not otherwise indicated. MESI model implies lockstep execution.

Thanks to my pipeline model design which moves most simulation to DBT compilation time rather than run time, and to my memory system model design which offloads most memory accesses by using L0 caches, simulating pipelines and cache coherence protocols did not add a significant overhead themselves, compared to the overhead of lockstep execution.

Chapter 4

A Practical Virtual Cache Coherence Protocol

With data gathered using TLB simulation, I determined that a practical cache coherence protocol based on virtual addresses is feasible. In this chapter, I describe the principles that guided the protocol design and the detailed design of the protocol. I demonstrate how I used the fast cycle-level binary translator R2VM discussed in Section 3.5 to simulate and evaluate the protocol. I also identify some important cases involving dirty or exclusive cache lines, that if not carefully considered, could lead to data being lost or corrupted, and present the adopted solutions to these challenges and potential alternative approaches.

4.1 Design Principles

The experiment in Section 3.3.1 shows considerable benefits in sharing address translation infrastructure. Analysis of synonym use patterns in Section 3.4 suggests most of them are read-only. Building on these observations, I determined that a practical virtual cache and associated cache coherence protocol requires the following properties.

Efficient in Performance, Area and Power

More specifically, the protocol should optimise the common case path, i.e. when the L1 cache hits, in terms of delay and power, and ideally minimise the area of logic in the vicinity of each core. The area of infrequently accessed SRAMs or area far away from the core and closer to the L2 cache is less of a concern, given that modern technology nodes are often restricted in power and heat density.

Virtual caches can achieve area and power saving by removing the need for a TLB during an L1 cache access. However, designs with virtual caches are non-trivial as the issues associated with homonyms and synonyms must be overcome, as described in Section 2.5. Unfortunately, many simple synonym avoidance schemes require significant performance to be sacrificed, weakening the potential advantages of virtual caches. More complex schemes that add to the cache hardware or require complex coherence protocols also risk adding to area and power requirements and again outweighing the gains of a virtual cache approach.

Most of the previous attempts to use the virtual cache try to avoid exposing the synonym issue to the cache coherence protocol. They either operate cache coherency entirely in physical address space or perform remapping before the request beforehand, so a unique address is used for coherency, and thus a reverse mapping structure is unavoidable. Therefore I concluded that a virtual cache coherence protocol needs to be co-designed with the virtual cache.

Compatible with Existing Programming Model

Portability of existing programs is of ultimate importance. This requires very little or no changes to existing programming models and the ISA. Userspace programs should be able to compile unmodified, or even better, run without the need for recompilation.

As RISC-V is used as the basis of the design, this means that the atomic operations provided by RISC-V A extension need to be supported. RISC-V A extension provides LR/SC instructions (also known as load-linked/store-conditional), and they are very frequently used in compare-and-swap loops in the userspace. Single-instruction atomics like `amoadd` are also very frequently used in userspace programs, and with the emergence of concurrency-safe programming languages like Rust their use are increasing.

I concluded that for this requirement to hold, the SWMR (single-writer, multiple-reader) invariant [109] needs to be kept. The SWMR invariant makes the cache coherence protocol more generic, supporting most memory models, including TSO which is frequently seen in workstation and server-grade systems. Without SWMR, it is difficult to support atomics and is impossible to support stronger memory models. Particularly, without SWMR it is hard to provide the progress guarantee, i.e. LR/SC code sequence will eventually succeed and the whole system can progress.

Non-SWMR approaches can also hurt performance significantly because atomic instructions are no longer local. For example, if the non-MSI approach proposed by Kaxiras as mentioned in Section 2.5.9 is to be modified to support atomics (the described approach only supports fence-based synchronisation), all atomic operations would involve a bus transaction that propagates to the LLC which serves as the point of synchronisation.

Requiring Minimal OS Involvement for Correctness

Some approaches described in Section 2.5 require a lot of OS modifications to function correctly. An ideal solution should be compatible with existing portable OS kernels such as Linux with relatively little modifications needed for correctness. Of course, it is acceptable that more extensive modifications may be needed to maximise efficiency or performance.

4.2 Imperfect Reverse Translation and Batch Invalidation

Using data gathered in Section 3.3.1 and Section 3.4 as a guidance, I designed a virtual cache coherence protocol based on "imperfect reverse translation and batch invalidation". In short, it is a modified version of the directory-based MESI cache coherence protocol with L2 backpointers. Each core has private L1 instruction and data caches, but they do not have their own TLBs or page table walkers. All L1 caches operate solely using virtual addresses. A single[1] TLB is shared between all cores and is located close to the L2 cache. The L2 cache is physically-indexed and tagged with both physical and virtual tags.

The typical L2 backpointer approach keeps a virtual tag in the L2 cache and forbids synonyms from occurring in L1 caches at all, thus excluding read-only synonyms. The decision rationale is that it is expensive, if not impossible, to keep track of the full backward translation if synonyms were to occur. My design simply accepts that full backward translation is infeasible. While efforts are taken to maintain a backward translation as accurate as possible, an imperfect reverse translation can be tolerated in this design if needed.

Mathematically, let P be addresses from the physical address space, and V be addresses from the virtual address space. To avoid homonym issues, assume all addresses in V are tagged with ASIDs. When coherence messages (e.g. cache line acquisition or writeback messages) flow from the L1 to the L2 cache, page tables and TLB(s) provide forward address translation from virtual to physical addresses. Let this partial function be $f : V \to P$. Whenever the L2 cache needs to invalidate a cache line from the L1, we need a reverse translation R from P to V.

If V_1 represent the set of all virtual addresses present in the L1 cache, then the "perfect" reverse translation R is

$$R_0 = \{(p \in P, v \in V) \mid v \in V_1 \land f(v) = p\}$$

[1]This protocol itself permits the use of multiple TLBs, including the instantiation of one private TLB, placed after the L1 cache, for each core. Naturally, the TLB could also be banked. I use a single TLB in the description and implementation for simplicity and to permit maximum sharing.

Due to the presence of synonyms ($\exists v \neq v'.\ f(v) = f(v')$), R_0 is only a binary relation but not a partial function (i.e. one PA can map to 0, 1 or more VAs). This creates difficulty in keeping and maintaining such structure in hardware.

However, it is *not necessary* to keep a perfect reverse translation. Let R be the reverse translation representing a structure maintained by hardware, we only need $R \supseteq R_0$ to enable proper back invalidation – excessive invalidation can reduce performance but it will not affect correctness. The proposed "imperfect reverse translation" exploits this property, relaxing the requirement on R instead of adding restrictions so that R_0 becomes a partial function, like synonym avoidance.

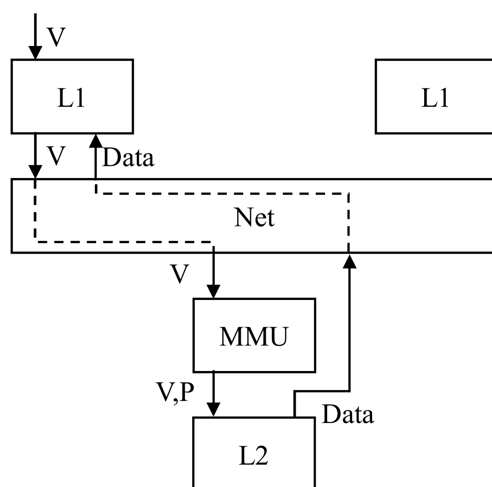
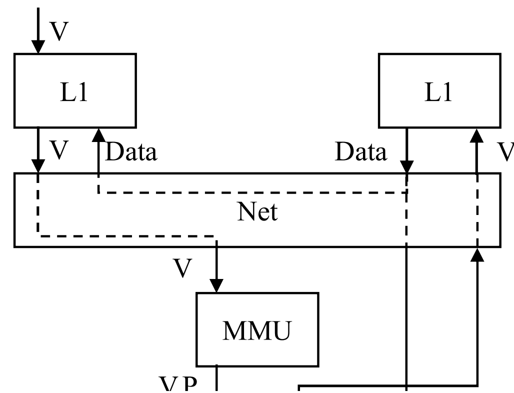


Figure 4.1 L1 miss flow when the cache line is neither owned nor shared

Figure 4.1 shows the message flow when L1 misses and the cache line is not owned or shared by any other L1 caches. As indicated in the figure, the interconnect that L1 caches connect to operates solely with virtual addresses. The single shared MMU, including a TLB and a PTW, translates virtual addresses to physical addresses, and both virtual and physical addresses get passed to the L2 cache. In this figure, the L2 cache responds directly with data, marks the requester as the owner of the cache line and records the virtual address used



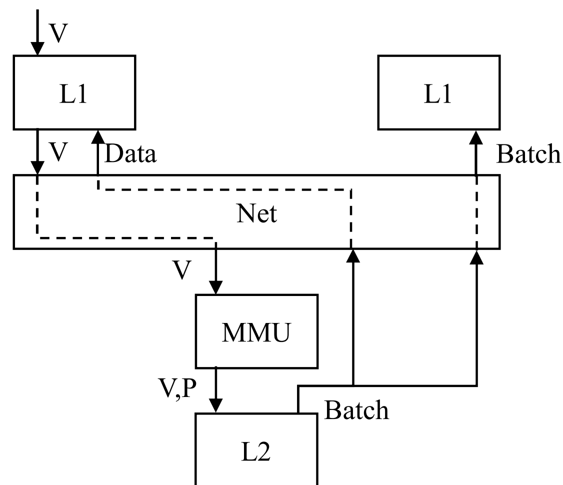


Figure 4.3 L1 miss flow when the cache line is shared with synonyms present. For simplicity the acknowledgement messages are omitted in the figure.

An L2 cache line will enter a synonym state when a read-only request to it carries a virtual address that differs from the one currently recorded at this cache line. In this state, no particular virtual addresses are associated with the cache line, so when invalidation is required, e.g. eviction or write is requested, we are unable to send correct invalidation message to the L1 like Figure 4.2 did. A batch invalidation message is sent instead to all sharers, as shown in Figure 4.3. The only address information carried by such a batch

invalidation is the page offset, and it could match to any number of L1 cache lines with the same page offset.

Batch invalidation will not be ideal performance-wise, and its only purpose is to ensure correctness. The assumption is that the number of such invalidation should be very small. As pointed out in Section 3.4, most synonyms are read-only, while a very limited number of applications will make use of read-write synonyms. I intend to keep the single-write-multiple-reader (SWMR) invariant for the proposed protocol, so read-write synonyms are not present. Therefore $r(p)$ can only be $\top$ is there is a read-only synonym, so batch invalidation should be infrequent. When a synonym is detected, a simple heuristic is used to distinguish if the synonym would be read-only or read-write, by checking the page property bits of the PTE used to access the cache line. The cache line will only change $r(p)$ to $\top$ if the writable bit is not set.

Table 4.1 shows the comparison between physical caches, various previous virtual cache approaches described in Section 2.5, and the proposed cache coherence protocol. The proposed solution removes components required on the critical path of L1 cache access. It permits read-only synonyms to co-exist in caches, and does not require software involvement to ensure correctness of operation even when synonyms. This is all achieved while maintaining compatibility with strong consistency models and atomic instructions. Like all other designs in computer architecture, this is not done without making trade-offs: they include mitigating instead of tolerating writable synonyms in caches, and occasionally requiring batch invalidation. Despite the trade-offs made, the protocol still satisfies all the design principles outlined in Section 4.1.

4.3 Simulation Technique

The initial protocol design consists of three channels, namely `Req`, `Resp` and `Wb`. `Req` and `Wb` channels flow from the host to the device, while the `Resp` channel flows in the other direction. There are 7 main types of messages:

- `Get/Upgrade`, on channel `Req`: Cache line acquisition or permission upgrade. Carries the virtual address, requested permission, current `SATP`, which consists of the ASID and address of the root page table in use.

- `GetAck`, on channel `Resp`: Acknowledgement to `Get/Upgrade`. Carries the granted permission, page property bits from page table, and contents of the cache line.

- `GetNack`, on channel `Resp`: Negative acknowledgement to `Get/Upgrade`. This is unique to my virtual cache coherence protocol design; it will be the response (instead

	Physical (PIPT)	Synonym Avoidance	VIPT	L2 backpointer	Hybrid	Write-through	Proposed
Read-only synonyms	Not an issue	Mitigated in SW	Allowed	Mitigated in HW	Assisted by SW	Allowed	Allowed
Read/write or write/write synonyms	Not an issue	Mitigated in SW	Allowed	Mitigated in HW	Assisted by SW	Allowed	Mitigated in HW
Supports strong consistency models	Yes	Yes	Yes	Yes	Yes	No	Yes
Supports atomics	Yes	Yes	Yes	Yes	Yes	No	Yes
Components needed for L1 access	TLB	None	None	None	TLB (Sometimes)	None	None
Extra invalidation complexity	None	None	Scan	None	None	Cache flush	Scan sometimes

Table 4.1 Comparison between physical (PIPT) caches and different virtual cache techniques

of `GetAck`) if the property bits of the PTE indicate that requested permission cannot be granted, e.g. requesting write permission to a cache line in a read-only page, or a request to a cache line in a non-present page.

- `Put`, on channel `Wb`: Voluntarily relinquish permission to a cache line. Carries the virtual address, downgraded permission, and the contents of the cache line.
- `Inv`, on channel `Resp`: Invalidation, involuntary permission downgrade initiated by the next-level cache. Carries the tagged virtual address, target permission, and the scope of invalidation (i.e. whether this is a wildcard or a specific invalidation).
- `InvAck`, on (

There are also mes

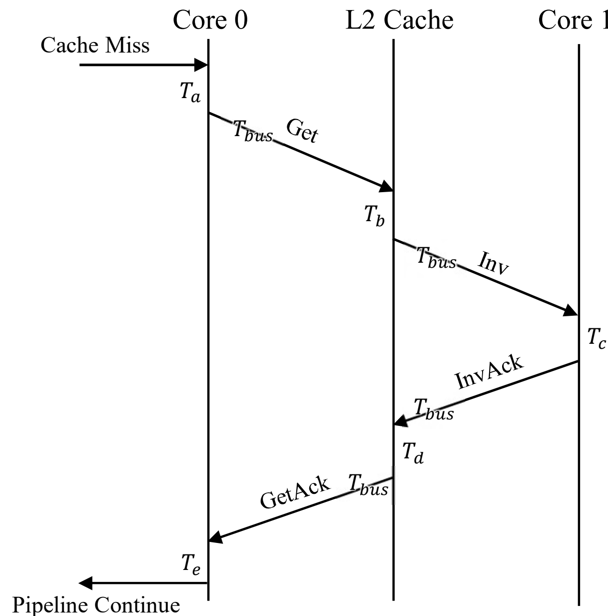


Figure 4.4 A typical flow of the cache coherence protocol being simulated

A typical flow involving invalidation is shown in Figure 4.4: T_a, T_b, T_c, T_d and T_e are the delays of components reacting the received event of message, respectively, where T_{bus} is the delay of the interconnect.

In R2VM, each core runs on its own fiber. While it is possible to introduce new fibers to simulate cache components like the L2 cache, doing so would impact performance significantly. The simulation code of L2 is ran on the same fiber as the L1 cache that initiates the request. Serialisation is achieved using mutual exclusion (mutex) instead. As depicted in Figure 4.5, when a cache miss happens, the L1 cache code will sleep for $T_a + T_{bus}$ cycles,

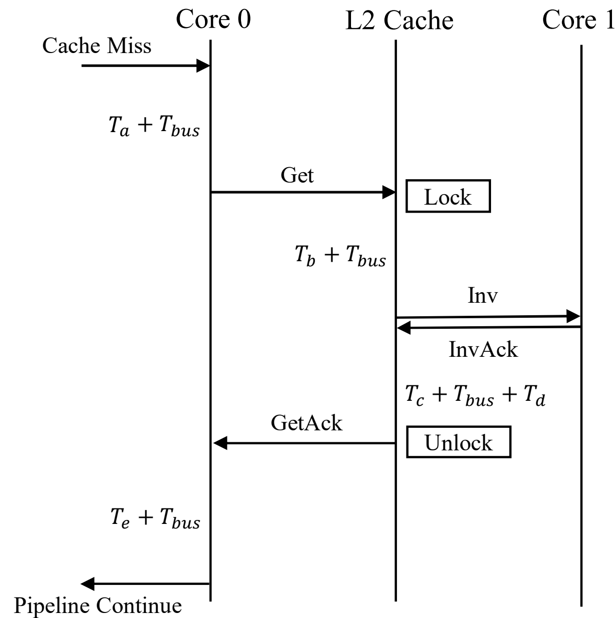


Figure 4.5 The same flow in the simulation implementation of the cache coherence protocol

accounting for its intrinsic delay and the delay needed for the message to reach L2 over the interconnect. The mutex is then locked, ensuring the L2 cache will not serve multiple other L1 caches simultaneously in the same cycle. The delay from the L2 to the L1 is simulated by sleeping after unlocking the mutex.

The invalidation part poses a challenge in R2VM simulation: as described in Section 3.5.3, it is difficult and impractical to interrupt the execution of a binary-translated program mid-way. Instead, the invalidation on another L1 cache is also performed on same fiber as the core that initiates the transaction asynchronously with compare-and-swap (CAS) loops. The delay of the L1 cache's invalidation operation thus has no effect to the invalidating core. This approximation is a source of inaccuracy; the initiating fiber will sleep for T_c cycles with mutex held, therefore the initiating core and other cores that initiate any transactions during these cycles will observe the effect of the delay, mitigating the inaccuracy.

4.4 Evaluation Technique

4.4.1 Baseline

The baseline for evaluation is a physical cache setup with a directory-based MESI cache coherence protocol. The baseline setup has 4 cores running in 100MHz[2], each with the following parameters:

- L1 instruction cache: 16KiB, 4-way associative, non-coherent
- L1 data cache: 16KiB, 4-way associative (note that this is the largest L1 size that still allows parallel TLB lookup with the given associativity, 4 * 4KiB = 16KiB).
- Instruction TLB: 32 entries, 4-way associative
- Data TLB: 32 entries, 4-way associative
- L2 cache: 256KiB shared by all cores, 4-way associative, inclusive of L1

The virtual cache protocol under evaluation has similar parameters:

- L1 instruction cache: 16KiB, 4-way associative, non-coherent
- L1 data cache: 16KiB, 4-way associative
- Unified TLB: 256 entries shared by all cores, 4-way associative
- L2 cache: 256KiB shared by all cores, 4-way associative, inclusive of L1

I consider this comparison fair because the total size of L1/L2 caches and the number of TLB entries are identical, and they share the same associativity. The difference between these two setups is the main objective for evaluation.

The delay and latency parameters are set to be mostly identical for both protocols. For the virtual cache protocol, 1 extra cycle delay is added for all communications between L1/L2 caches to account for the TLB access (for physical cache the TLB access is parallel to the cache access). Batch invalidation will also take extra cycles for the L1, with the cycle count equal to the number of sets that can potentially contain the synonym. This accounts for the scanning required by the hardware when the L1 cache handles a batch invalidation.

[2]The selected simulation frequency determines the number of instructions that are executed between timer interrupts. The number is chosen to be of the same magnitude as possible frequencies of a FPGA implementation.

4.4.2 Benchmarks

PARSEC benchmark, as described in Section 2.8, is used for evaluation of the general performance. PARSEC benchmarks are multithreaded; this means while they can be helpful to evaluate performance differences between the physical cache baseline and the setup with the virtual cache protocol, they will not create synonyms and therefore cannot be used to evaluate the performance characteristics of the protocol in presence of synonyms. As discussed in Section 3.4, synonyms arise from shared memory mappings in different processes. In practice this happens commonly with multitasking, and I am not aware of existing well-established benchmarks that can stress this scenario. I therefore developed a set of microbenchmarks, `minibench`, to evaluate the performance of the cache coherence protocol in those specific cases.

Minibench will create 3 threads, with one managing thread and two runner threads. All threads are pinned to distinct cores, preventing interference from the kernel scheduler. The managing thread first sets up the runner threads, and waits until they are ready. When each runner thread has started, it will signal the managing thread and busy spin on an atomic variable. The managing thread will start all runners at once when all of them signal their readiness. After a fixed test period lapses, the managing thread will signal the runner threads to stop, wait for them to finish and aggregate the results.

Minibench has 3 modes of thread organisation:

- `thread`: Represents typical multithreaded applications. The runner threads will be within the same process. In this case, there is only a single address space so no synonym is present. Memory is mapped using a shared read-only mapping.

- `fork`: Represents multiprocessing applications with forking. The runner threads will be the only threads of forked processes. In this case, there are multiple address spaces but virtual addresses will be identical, so there will be different-ASID same-address synonyms. Memory is mapped using a shared read-write mapping.

- `multiprocess`: Represents multiprocessing applications with spawning. The intention of this mode is to model two independently spawned processes (each with one single thread) with shared memory, e.g. the scenario when using Python's `multiprocessing` library. In this case, there are multiple address spaces and the virtual addresses for the shared memory are not guaranteed to be identical, so there can be different-ASID different-address synonyms. Implementation-wise, to ensure that the virtual addresses are indeed different (rather than relying on the ASLR), minibench does not spawn processes independently. Instead, it will map a same file twice in the same process,

fork, and then instruct each runner thread (process) to use different mappings. Memory is mapped using a shared read-write mapping.

Minibench contains three microbenchmarks:

- `scan`: Linearly reading from memory with fixed stride. Various mapping sizes are tested with a stride of 64 bytes (cache line size), including 4096 (smaller than the L1 cache size), 32768 (larger than the L1 cache size), 524288 (larger than the L2 cache size). Mapping of size 2097152 with a stride of 4096 bytes (page size) is also tested.
- `modify`: Linearly modifying memory using atomic fetch-and-add instruction, with fixed stride. Sizes and strides tested are identical to `scan`.
- `sync`: Acquiring and releasing two semaphores alternatives to synchronize between two threads.

The `scan` benchmark is the main performance indicator, as it reflects the most common synonym scenario. Other configurations do not guide the design of the protocol, but they do provide useful insights. For example, the `modify` workload can represent the read-write synonym scenario like the Wayland shared memory buffer as discussed in Section 3.4.

4.5 Dirty Cache Lines and Page Table Entry Changes

One critical property of a cache coherence protocol is that when the L1 requests a cache line and modifies it, the dirty content must be written back to the same physical address that it retrieves the initial data from. This creates a challenge for the proposed protocol: as bus requests sent from L1 caches use virtual addresses but the L2 is indexed by physical addresses, all messages from the L1 to the L2 must pass through the shared TLB. RISC-V specification only requires an `SFENCE.VMA` instruction *after* changes are made to the page table; if the address translation changes between a read and a writeback of a cache line, the dirty data could be written to a wrong physical address. The issue is further aggravated by the use of exclusive (E) state in the MESI protocol; the same issue also applies to cache lines in E state because the L1 cache can modify and upgrade them to M state unilaterally, without bus transactions.

This is a non-issue for physical caches, so the simplest and the most straightforward option, is to keep a copy of physical addresses in the L1 cache metadata, and bypass the TLB for dirty data write-back. This is optimal in terms of performance, requires no ISA or OS changes, but it requires adding many bits to the L1 cache where the area is precious, and it

defeats the purpose of removing the TLB from the vicinity of the L1 cache in the first place, therefore I considered this approach as unacceptable for practical implementation. Given its performance advantage I still ran experiments with this approach and used it as the evaluation baseline in this section.

Another approach would be to guarantee that the original translation stays in the TLB. One way to enforce that is to make the shared TLB inclusive of the L1 cache – whenever an entry is evicted from the TLB, all L1s must invalidate and write-back all their entries related to the evicting page. However, the performance impact of this approach is significant: TLB eviction is not a rare event, and the cache flush induced by the TLB eviction requires a search through the cache and therefore is time-consuming. As shown in Figure 4.6, for the PARSEC benchmark suite, the average performance of forcing the TLB to be inclusive is 12.3% slower than the baseline, and for the two benchmarks with high TLB miss rates, `blackscholes` and `canneal`, the performance is reduced by 46.0% and 41.4% respectively.

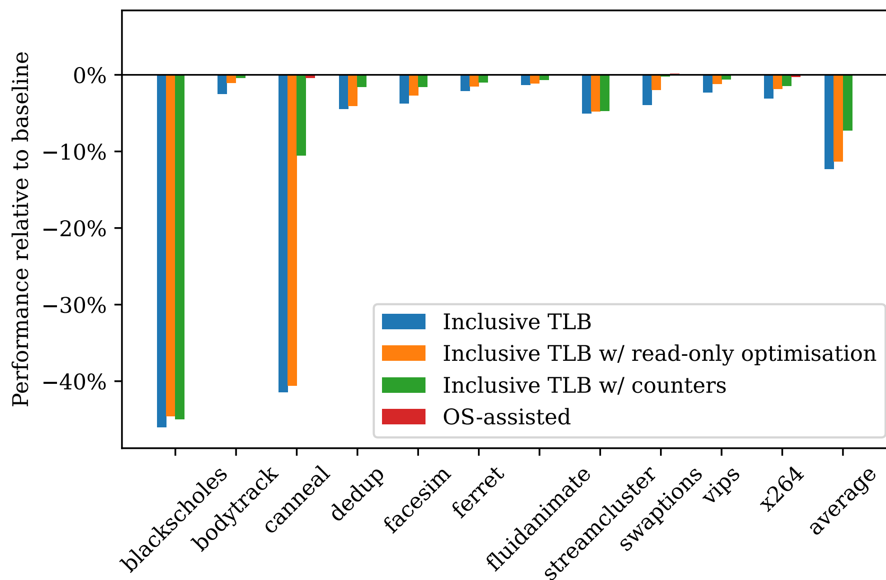


Figure 4.6 PARSEC benchmark performance of evaluated PTE change options, relative to storing physical address in L1

One optimisation that can be made to the inclusive TLB approach is to use the page property bits of the evicted TLB entry to check if a dirty cache line within this page can exist. If either the W (writable) bit or the D (dirty) bit is clear, then no dirty cache lines related to this page can exist. After applying this optimisation (labelled “w/ read-only optimisation” in Figure 4.6), the performance penalty drops only slightly to 11.3% for PARSEC average, 44.6% for `blackscholes` and 40.6% for `canneal`; this is still too high to be acceptable.

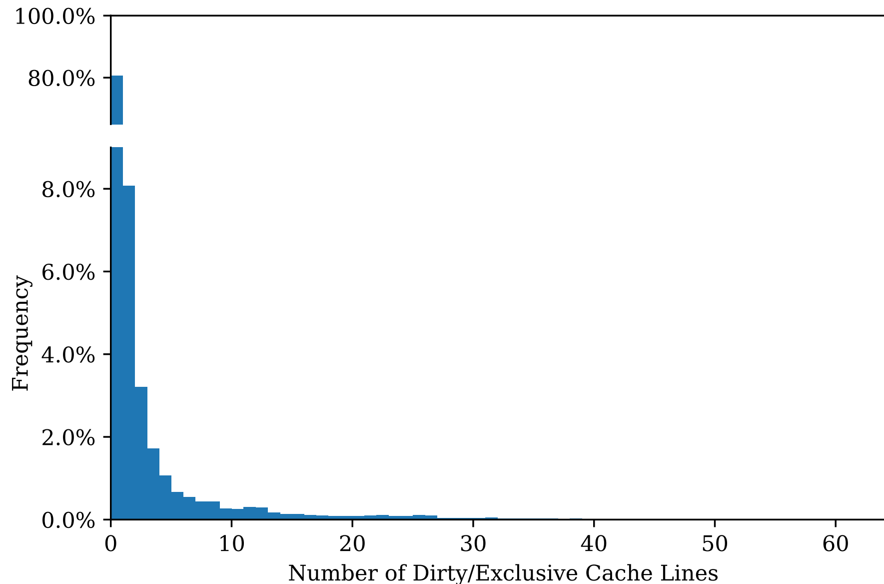


Figure 4.7 Number of dirty/exclusive cache lines related to a page present in the L1 when the page is evicted from TLB

I instrumented the simulator to generate a histogram that shows the number of writable cache lines that corresponds to the page when evicting a writable page from the shared TLB. The result collected from running canneal is shown in Figure 4.7. The result shows that the majority (81%) of the pages, when evicted, have no dirty/exclusive cache lines present in the L1. I therefore also experimented with the idea of adding a counter to each TLB entry. The counter is incremented whenever a cache line is granted to the L1 cache with write access, and subtracted when the cache is invalidated or voluntarily released from the L1 cache. If the counter of a particular page is 0, then no dirty cache lines related to this page can exist in L1 caches, and therefore L1 invalidations can be omitted when evicting this page.

This optimisation (labelled "w/ counters" in Figure 4.6) delivers some gains: `canneal` is now only 10.5% slower than the baseline, and the average performance is 7.3% slower than the baseline. Despite the huge performance improvement over the naive inclusive TLB approach, 7.3% slowdown is still very significant and therefore I concluded that an inclusive TLB approach was not feasible.

The final design decision taken was to keep the TLB non-inclusive. Instead of ensuring the consistency of address translations in hardware, the responsibility is shifted to the OS. Fortunately, the required hardware and software changes are small and easy to make. For the hardware, only a cache flush instruction is needed. For the OS, instead of performing a PTE update followed by a TLB flush, the OS instead does a cache flush first, updates the

PTE, and then performs a TLB flush. The Linux kernel already has support for this, so the only modifications needed for the kernel are implementing the cache flush functions with the added instruction and switching them on. As the frequency of page table modifications is very low (and already expensive), the additionally required cache flush operations should have a minimal impact on performance. This is confirmed by simulation results, showing only an average of 0.07% slowdown compared to the optimal baseline which stores copies of physical addresses in the L1. This approach is 7.8% faster in average compared to the counter approach.

4.6 Dirty Cache Lines and ASID Changes

The previous section discussed how PTE changes affect the dirty cache lines and my solution to the problem. There is still a challenge that remains. In unmodified RISC-V systems, each hardware thread will store the ASID and associated root page table number (and current ASID) in its `SATP` CSR. When the OS context-switches from one process to another, it will re-set the `SATP` CSR. When there is a dirty cache line to be written back, the only information that is stored in the L1 cache is its virtual address and corresponding ASID. If the ASID is not the currently active one specified in the `SATP` CSR, then if the TLB misses, there is no way to redo the page table walk because the root page table number no longer exists in a CSR.

I devised two methods to address this issue:

1. Flush dirty, non-global cache lines from L1 caches when ASIDs change.

2. Keep a global, shared, table of root page numbers (one extra level of indirection). Instead of storing root page table numbers in CSRs, we can add an extra level of indirection by creating a table of root page table numbers. The PTW of the shared TLB can retrieve the root page table number of a specific ASID from this table before proceeding to ordinary page table walking.

Method 1 is simple and requires no OS change; only a minor cache modification is needed to perform a cache flush whenever the `SATP` CSR is changed. However, this method prevents the implementation of a shared virtual L2 cache (with the L3 cache being physical and a TLB between the L2 and the L3) which the original protocol allows. Option 2 requires OS changes to maintain the data structure needed for the extra indirection, but it permits a shared virtual L2, allows the bus transactions to no longer include the full root page table number but merely the ASID, and can potentially have better context switch performance as the cache flush is not needed.

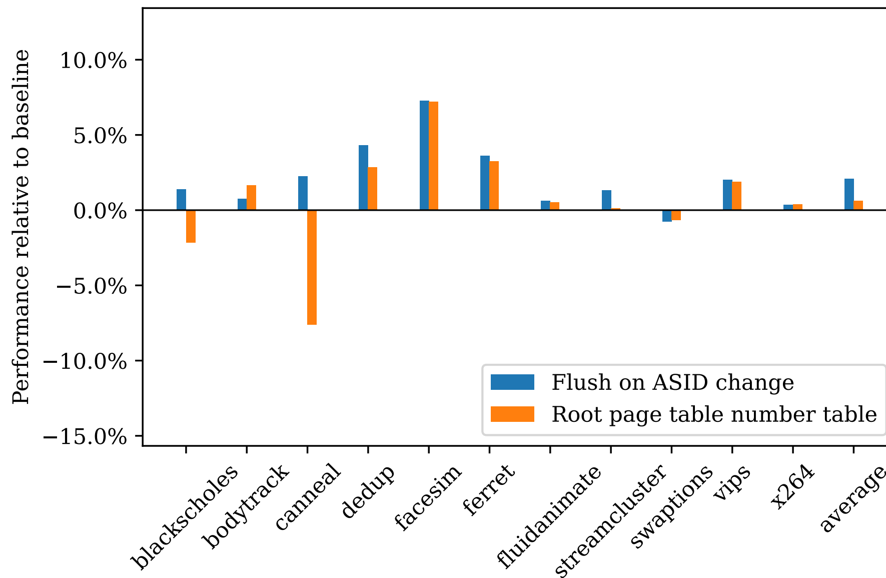


Figure 4.8 PARSEC benchmark performance of evaluated ASID change options, relative to physical cache baseline

Upon evaluation, it turns out that the performance penalty of a cache flush when the ASID changes is not significant. An ASID change indicates a process change; if the working set of the newly scheduled program is larger than the L1 cache size (and this is the case for many programs), then those dirty cache lines would eventually be evicted and written back anyway, so flushing them ahead of time is not a huge performance penalty; the cache lines used by the kernel or the firmware are global or physical, and they do not need to be flushed[3]. On the other hand, option 2 has a drawback that the extra level of indirection will slow down the page table walking due to an extra memory access; this is reflected in the performance of PARSEC runs in Figure 4.8. There are performance penalties in all workloads, with `blackscholes` and `canneal` again being the most significant ones. Option 2 is 1.5% slower than option 1 on average. I therefore decided to use method 1 for its simplicity and slight performance advantage.

4.7 L2 Eviction of Read-only Synonyms

When running PARSEC benchmarks, synonyms are usually not encountered. This section considers the additional issues that are present when synonyms are possible.

[3]This assertion is only true when all global pages are synchronised. It is later discovered that the assumption does not hold in Linux, and Section 5.3.2 outlines the solution that mitigates the issue.

In my original design, batch invalidation exists to ensure the correctness for a rare case when a write is attempted at an established read-only synonym. In the simulation, it is discovered that in addition to writes, evicting these read-only synonym entries may also require the use of batch invalidations.

The L2 cache has limited size, and will need to occasionally evict cache lines, but in my mathematical formulation, the domain of r is V, the entire virtual address space. Therefore, for any physical cache line p not present in the L2, it needs to use a default value for $r(p)$. The natural value of this default value is $\bot$, i.e. it is not active in any L1s. For soundness, an L2 eviction would need to reset a cache line to this state, so a batch invalidation is required to evict a read-only synonym from the L2.

One approach would be to select $r(p) = \top$ as the default state, then evicting L2 entries would be free. However doing so means that a batch invalidation is needed whenever a new cache line is pulled from the next-level cache into the L2 for write accesses. This would result in excessive batch invalidation even for cache lines that are not read-only synonyms, and therefore is ruled out.

I have come up with various approaches that could reduce the frequency of needing batch invalidations.

It should be noted that the cost of performing batch invalidation varies depending on the design of the L1 cache; for example, for a 4-way associative 16KiB L1 cache, if the cache uses the least significant bits of the address directly as the set index into each way, then the cost of batch invalidation is as cheap as a normal invalidation because the least significant 6 bits correspond to the cache line offset within a page and is not relevant to the virtual address. If the cache is larger, or uses a hash function to map the address to a set index, then the cost of batch invalidation is more expensive because it requires scanning through multiple sets to find the matching cache lines. As I intend to design the protocol to be generic and agnostic to the exact design of L1 caches, for the evaluation in this section I prevent the L1 cache from reducing sets to scan by looking at the least significant 6 bits.

4.7.1 ASID Batch Invalidation

The first approach aims to reduce the cost of batch invalidation. A full batch invalidation will require the L1 to scan all its entries, therefore the high cost. However, as discussed in Section 3.4, one of the major sources of read-only synonyms is from `fork` system call in Linux. These synonyms have a unique property that their virtual addresses are always identical. These are synonyms purely because they are tagged with the different ASIDs.

It is therefore a waste to set $r(p) = \top$ in this case. Let $V = A \times U$ where A is the set of all ASIDs and U the set of all untagged virtual addresses. I redefine $r(p) : P \to$

$(A \cup \{\top\}) \times U \cup \{\bot, \top\}$. Compared to the old $r(p)$ definition it now allows cache lines to have a reverse translation state of $(\top, u)$, which means that when doing invalidation, ASID is wildcard matched but the virtual address is exactly matched. This invalidation is still "batch" because it can invalidate multiple cache lines, but it does not require a scan through all cache lines, merely cache lines in a single associative set, greatly reducing the performance penalty.

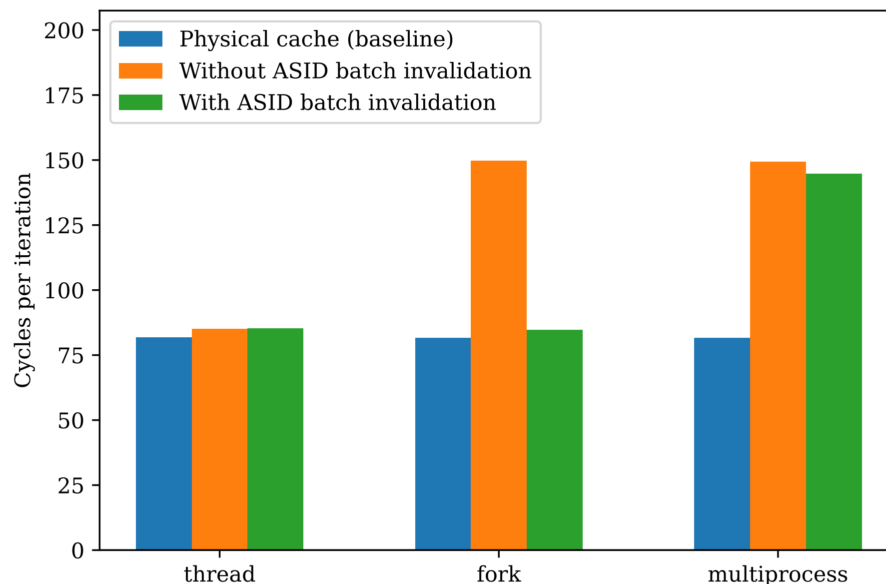


Figure 4.9 Minibench evaluation of ASID batch invalidation, with a `scan` run, size of 524288 bytes, stride of 64 bytes

Figure 4.9 demonstrates the performance metrics of this change. Minibench is used for evaluating this change, with the size 524288 chosen because it is larger than the 256KiB total L2 size, therefore hitting the L2 eviction scenario. It can be seen that in `fork` mode, with partial batch invalidation applied, the virtual cache has similar performance as the physical cache, while the performance suffers if partial batch invalidation is not implemented. The effect is not as dramatic for the `multiprocess` scenario, as the synonyms are "full" synonyms where the virtual addresses actually differ rather than just the ASIDs. Both virtual cache implementations have significant performance penalty compared to the physical cache in this scenario.

4.7.2 OS Assisted Synonym Placement

Synonyms that are not a result of a `fork` would not benefit from the aforementioned hardware optimisation automatically. However, we could tweak the OS so that it tries to map synonyms at the same virtual address (not necessarily the same ASID). This is possible in 64-bit systems

in which virtual address space is significantly larger than the physical address space. If this is done, then the ASID batch invalidation mentioned above would be used instead a full batch invalidation, improving the performance.

Haria et al.'s devirtualised memory [60] addresses synonyms in a similar idea by requiring synonyms to be devirtualised, i.e. have a PA=VA mapping. However, this requires memory regions with synonyms to be placed contiguously in physical memory, which may be difficult. In my design this is not needed, the OS can still map virtual addresses with synonyms to any physical addresses. The only assistance needed from the OS is to attempt to map synonyms in the same virtual address in the two distinct address spaces.

Hajinazar et al.'s virtual block interface [57] uses a single virtual address space for all processes, with different processes having different permissions to each page, and disallow synonyms. In spirit, the OS-assisted synonym placement is similar to Hajinazar's design, because synonyms are mapped in the same virtual address, but unlike Hajinazar's design, the proposed protocol is less intrusive, introduces no external fragmentation, and does not require all programs to be position-independent code.

There is a huge drawback with this approach, however. It nullifies ASLR: because shared libraries would have to be mapped to the same virtual address space, an attacker would know where the `libc` is mapped on all other addresses by simply looking at its own. This would make return-to-libc attacks [38] easy to achieve.

4.7.3 Partial Batch Invalidation

Even with ASLR enabled and shared mappings that would result in synonyms with different virtual addresses, optimisations are possible if the mappings are co-aligned to a value larger than a page size. For example, if process A maps libc to 0x7fbd801b7000, and process B maps libc to 0x7f9729e97000, then the least significant 17 bits are both 0x17000. If L1 caches are n-way set-associative, this means that they will be mapped to the same associative set if the L1 cache is smaller than $n\times$128KiB, which is almost always the case. A small co-alignment can still be helpful: if only the least significant 13 bits match, optimisation is still attainable – the L1 only have to scan half as many entries for a batch invalidation.

The information can be easily represented using bit masks, as shown in Figure 4.10. When a synonym is to be introduced, the L2 cache can compare the existing address and masks with the new address and produce a new mask. This mask is sent to the L1 cache when batch invalidation happens, and the L1 cache can use the mask to optimise the batch invalidation process. Definition of $r(p)$ in this case is changed to $r : P \to (V,V) \cup \{\bot\}$ and $R = \{(p \in P, v \in V) \mid \exists a,b.r(p) = (a,b) \wedge v \text{ AND } b = a \text{ AND } b\}$. The former $\top$ is now encoded as $(0,0)$.

ASID_LEN	VA_LEN-12
ASID	Virtual Page Number
111…111	1111…1111

No Synonym

X	Virtual Page Number
0	1111…1111

ASID Synonym

X	Partial VPN
0	111…111

Partial Synonym

X
0

Full Synonym

	ASID_LEN	VA_LEN-12
1	ASID	Virtual Page Number

No Synonym

0	1	Virtual Page Number

ASID Synonym

0	1	Partial VPN

Partial Synonym

0	1

Full Synonym

Figure 4.11 Encoding of different types of synonyms in partial batch invalidation

Of course, storing bit masks would incur a significant area overhead. To mitigate this issue, I introduced an encoding mechanism to encode the same information with only a single additional bit, as shown in Figure 4.11. Both the bit mask and the encoded format allow arbitrary number of significant bits, so the protocol can be generic and agnostic to the design of the L1 cache, unlike VIPT cache designs.

Apart from naturally occurring co-alignments, the OS can assist this process. For Linux, there is already a mechanism to do this, originally intended to guarantee the correctness of synonyms in VIPT caches. The same mechanism can be used to raise the co-alignment (except for the proposed protocol, higher co-alignment is an optimisation instead of a requirement for correctness, so it would not affect the portability of programs).

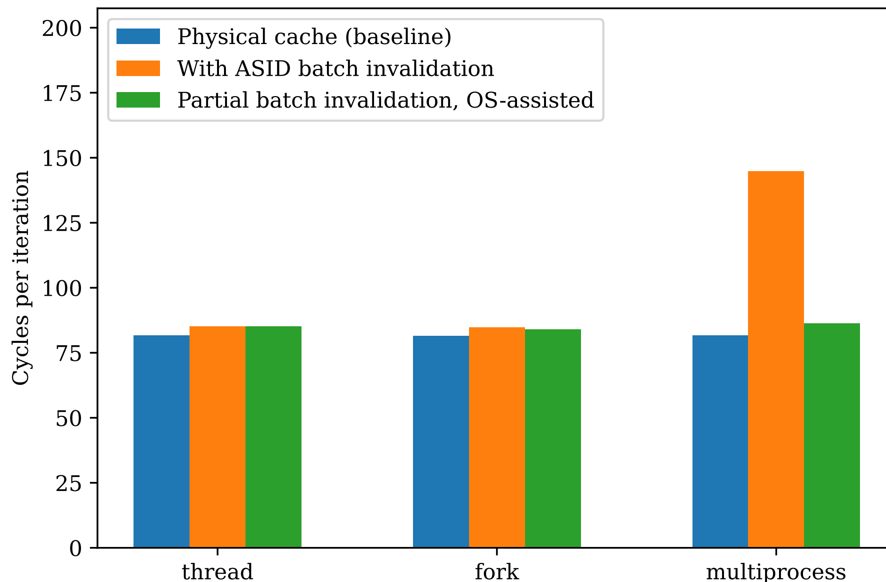


Figure 4.12 Minibench evaluation of partial batch invalidation, with a `scan` run, size of 524288 bytes, stride of 64 bytes

The partial batch invalidation technique is evaluated using the same minibench configuration used to evaluate the ASID batch invalidation approach. It can be seen on Figure 4.12 that the performance in a multiprocess synonym scenario improves significantly over just using the ASID-based batch invalidation optimisation, and is only marginally slower than the physical cache baseline.

4.7.4 Bloom Filter

One important observation is that the default state for $r(p)$ does not have to be either/or: whether it is $\bot$ or $\top$ can depend on the value of p. Mathematically speaking, if $r' : P \nrightarrow$

$V \cup \{\perp, \top\}$ be the states stored in the metadata of the L2, $f(p) : P \to \{\perp, \top\}$ be the default state, and

$$r(p) = \begin{cases} f(p) & \text{if } r'(p) \uparrow \\ r'(p) & \text{otherwise} \end{cases}$$

f does not have to be a constant function. Still, we cannot use many bits to represent such an f. A Bloom filter, being a probabilistic data structure, is therefore considered as a potential good choice for such an f.

Whenever a cache line in wildcard state is evicted, it is added into the Bloom filter. When a cache line is loaded from the next-level cache, the Bloom filter is looked up for its L2 state. Because the Bloom filter never gives a false negative, if $f(p) = \perp$, then it has not been inserted into the Bloom filter. If may give a false positive, i.e. $f(p) = \top$ for some p that has not been inserted into the filter; such scenario may cause unnecessary invalidation but nonetheless is not a correctness issue, as the resulting $r(p)$ and associated R satisfy the required property of $R \supseteq R_0$. The Bloom filter is cleared whenever a full batch invalidation occurs.

Bloom filter has two parameters, including m total number of bits and k number of hash functions. I performed theoretical analysis to choose parameters before implementing them in simulation and hardware. Assuming the ratio of synonyms in the working set is φ, and all accesses have independent and equal probability of being a synonym. For a given Bloom filter with n cache line tags inserted, we then have the false positive probability

$$p(n) = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k}$$

We can model the expected number of cache eviction/refills before a full batch invalidation needs to take place as a random process:

```
def S():
    n = 0
    ret = 0
    while True:
        # Refill a non-synonym cache line
        if random.random() < p(n) and random.random() > phi:
            # Full batch invalidation
            return ret

        # Eviction
```

```
    if random.random() < phi:
        # Synonym, insert into Bloom filter
        n += 1

    ret += 1
```

it can be rewritten into mathematical and recursive form as

$$S(n) = \begin{cases} 0 & \text{if } A < p(n)(1-\varphi) \\ S(n)+1 & \text{if } A > p(n)(1-\varphi) \wedge B > \varphi \\ S(n+1)+1 & \text{if } A > p(n)(1-\varphi) \wedge B < \varphi \end{cases}$$

where A, B are two independent random variables uniformly distributed in $[0,1)$. The expectation therefore is

$$\begin{aligned} \mathbb{E}[S(n)] =& p(n)(1-\varphi)\cdot 0 \\ &+ (1-p(n)(1-\varphi))(1-\varphi)(\mathbb{E}[S(n)]+1) \\ &+ (1-p(n)(1-\varphi))\,\varphi\,(\mathbb{E}[S(n+1)]+1) \end{aligned}$$

which reduces to

$$\mathbb{E}[S(n)] = \frac{1-p(n)(1-\varphi)}{1-(1-p(n)(1-\varphi))(1-\varphi)}(1+\varphi\mathbb{E}[S(n+1)])$$

let $a_n = \frac{1-p(n)(1-\varphi)}{1-(1-p(n)(1-\varphi))(1-\varphi)}$, then $\mathbb{E}[S] = \sum_{i=0}^{\infty}\left(r^i \prod_{j=0}^{i} a_j\right)$.

This analytical solution precisely matches the result of sampling the random process or running a simulation with a Bloom filter implemented, as indicated in Figure 4.13. Note that theoretically, usage of a Bloom filter should always be an improvement over not using it, since at least 1 synonym cache line needs to be evicted before the Bloom filter can mispredict (empty Bloom filter will always report $\perp$). This is reflected in the figure; if no Bloom filters are used, the number of accesses to invalidation follows geometric distribution and therefore has a mean of $\frac{1}{\varphi}$. The figure shows regardless of the parameter k chosen, the mean number will exceed $10 = \frac{1}{\varphi}$.

I have preselected reasonable parameters depending on feasibility and cost to implement in hardware. The number of bits m must be a power of two, so the hash functions can produce $\log_2(m)$ bits directly used as indices. Generally, the larger the m, the lower the false positive rate, but excessive size can be expensive to implement on hardware, especially on FPGA,

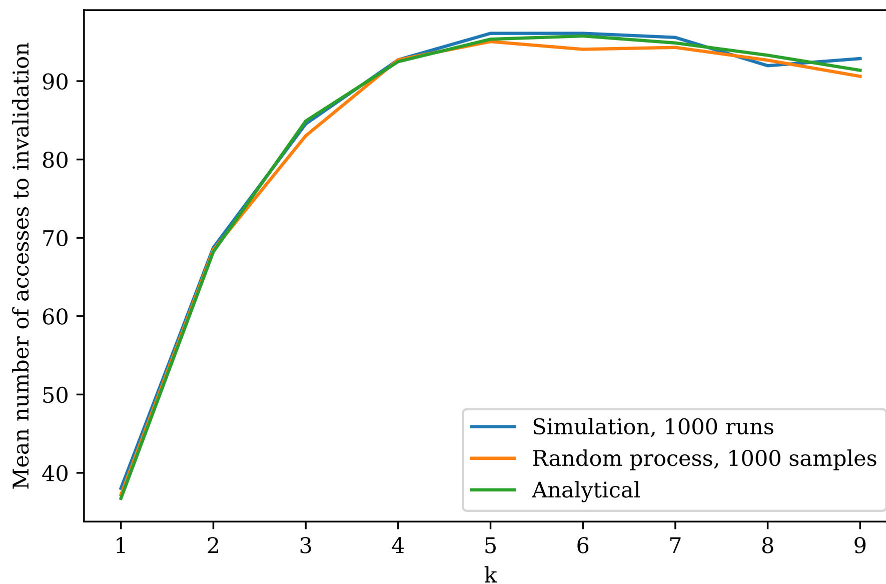


Figure 4.13 Comparison of results from simulation, random process sampling and analytical solution of the mean number of accesses to an invalidation for Bloom filter usage, with $\varphi = 0.1$ and $m = 64$

so I only selected $m \in \{32, 64\}$ for evaluation. For k, I choose $k \in [1, 6]$ to limit the total number of bits that hash function needs to produce.

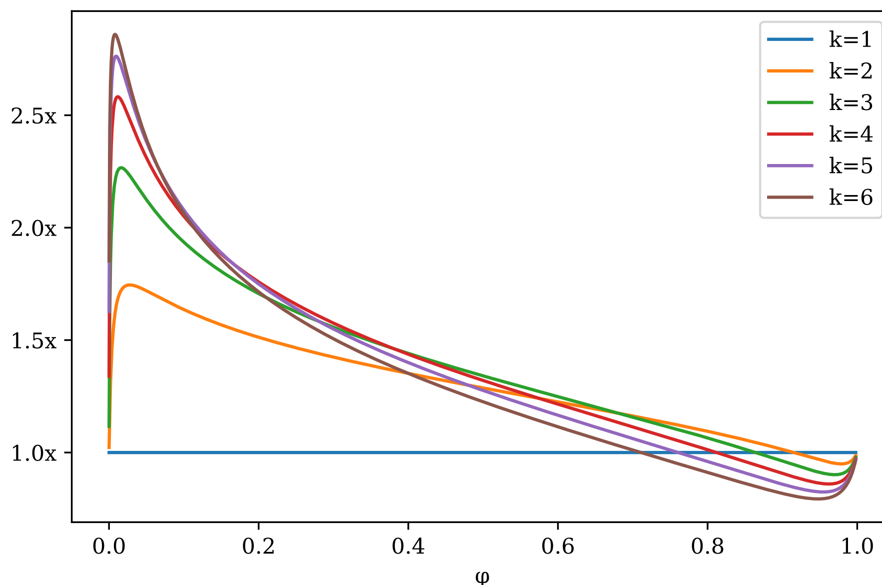


Figure 4.14 Mean number of accesses to an invalidation for $m = 32$, relative to $k = 1$, $m = 32$

The analytical solution is very helpful for selecting the optimal parameters from these values. From Figure 4.14 I decided that for $m = 32$, a good choice for k is 4. $k = 5$ only have minor improvement over $k = 4$ for small φ, at the expense of performance when $\varphi > 0.2$ and extra hardware implementation cost. Similarly, $k = 4$ also turns out to be a good choice for $m = 64$. $m = 64, k = 4$ always perform better than $m = 32, k = 4$ regardless of φ, so I choose that for evaluation in simulation.

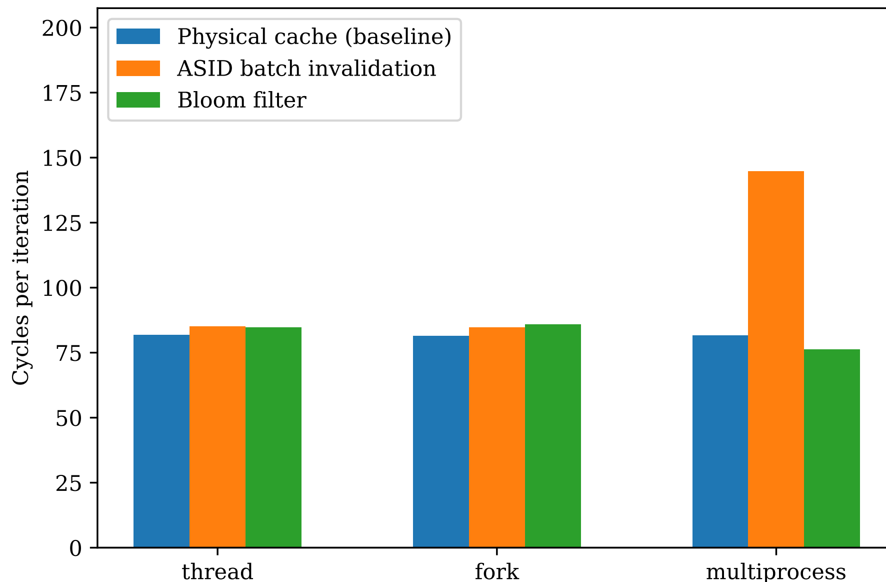


Figure 4.15 Minibench evaluation of the Bloom filter optimisation, with a `scan` run, size of 524288 bytes, stride of 64 bytes

Figure 4.15 shows the performance of the same minibench configuration. Interestingly, with a Bloom filter, multiprocess performance even exceeds the thread/fork scenario. This can be explained due to the overall reduction of invalidations, as eviction of full synonyms will only result them being added to the Bloom filter without even notifying the L1 caches.

Figure 4.16 shows the PARSEC performance comparison of the evaluated options described in this section. Because PARSEC is a multi-threaded benchmark and therefore rarely exercises the synonym eviction scenario, performance of all evaluated options have little difference. Average difference between any two options is within 0.4%.

Despite the apparent good performance with Bloom filter, I decided to implement the partial batch invalidation approach. There are a few reasons behind this decision:

- Partial batch invalidation is deterministic, compared to the probabilistic nature of using Bloom filters.

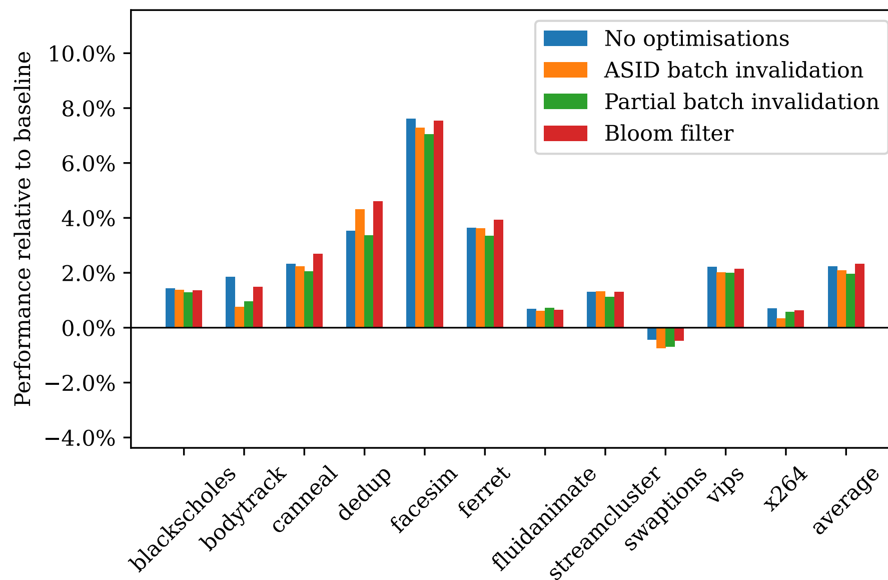


Figure 4.16 PARSEC benchmark performance of evaluated L2 synonym eviction options, relative to physical cache baseline

Figure 4.17 Metadata layout of an L1 cache line

The metadata stored in the L1 is shown in Figure 4.17. Because TLBs are not ever looked up for L1 cache access, page property bits in the page table need to be copied into the L1. In

PA_LEN-SETS_LEN-6		ASID_LEN	VA_LEN-12			
Physical Tag	S	ASID	Virtual Page Number	W	State	Sharer/Owner

Figure 4.18 Metadata layout of an L2 cache line

practice this is merely two additional bits (`U` bit denoting user/supervisor mode pages and a `G` bit denoting global pages). The readable/writable bits are checked when accessing the L2 and are reflected in the state of the cache line. Technically we also need a `P` bit to support physical addressing for machine mode or before supervisor mode code turns on paging, but since the ASID bits are not used for a global page, we set the `G` bit and use one bit from the ASID bits to distinguish physical addressing mode and a global page.

The metadata stored in the L2 is shown in Figure 4.18. Because L2 also acts as the directory, it contains the sharer/owner information about this cache line. As mentioned earlier, it also needs to represent $r(p)$, which we represent using an ASID, a virtual page number, and an extra bit to allow representation of synonyms, as shown in Figure 4.11. A `W` bit stores whether the cache line is previously accessed with write permission in the PTE; this is to facilitate the heuristics used for synonym read-only/read-write characterisation as mentioned in Section 4.2.

If an incoming request has address (a_i, u_i), the state transition algorithm for the L2 cache is:

- If the PTE is invalid, or a write access is requested but the page property is not writable, reject with `GetNack`. Otherwise it is translated to a physical address, say p.
- If the cache line is not owned or shared by any clients (L1 caches), set $r(p) = (a_i, u_i)$ and grant. A read request can be upgraded to a write grant if the page is writable (exclusive state).
- If the cache line is owned or the requested state is M, (batch-)invalidate the cache line, set $r(p) = (a_i, u_i)$ and grant. This rule ensures the SWMR property.
- Now the cache line is shared. We first compute the $r(p)$ if we simply grant the request. Let the current $r(p)$ be r, define

$$r' = \begin{cases} \top & \text{if } r = \top \\ \top & \text{if } \exists a, u.\ r = (a, u) \wedge u \neq u_i \\ (\top, u_i) & \text{if } \exists a.\ r = (a, u_i) \wedge a \neq a_i \\ (a_i, u_i) & \text{if } r = (a_i, u_i) \end{cases}$$

The above rule has ASID batch invalidation optimisation applied; the rule is more complicated and involves some bitwise operations with partial batch invalidation optimisation.

Now if $r = r'$, we just grant. The `W` bit is set if the page is writable.

- When $r \neq r'$, this means that we might be introducing or expanding the synonym state. For read-write synonyms, cache lines may be read-accessed before write accesses, so here heuristics come into play, to prevent a read-write synonym from being moved into a synonym state and therefore minimise the number of batch invalidation.

  We use the page's writable bit as a hint. If the page is writable, then we instead invalidate immediately, set $r(p) = (a_i, u_i)$ and grant. Similarly, invalidation will happen immediately if the `W` bit is set, this is to prevent accidental synonym state introduction when a page is mapped as read-only by one address space and read-write in another. One scenario for this to happen is a file-backed page can be mapped as read-only in one process, but is mapped as read-write in the kernel space. If the kernel just finished writing to the page, we would want to prevent introduction of synonyms if the page is next accessed from the userspace.

  If the page is neither writable, nor is `W` bit set, then we have high confidence that this is indeed a read-only synonym, so we can move into wildcard state by setting $r(p) = r'$ and grant without sending invalidations.

Figure 4.19 shows the summarised PARSEC performance characteristics of the proposed protocol. It can be seen that, on average, the proposed virtual cache coherence protocol is 2.1% faster than the baseline.

The TLBs' associativity and size can have significant performance impacts, so I also included variants of the setups in the evaluation for insights. For physical cache setup, I evaluated a variant with both the instruction and data TLB configured to be fully associative instead of 4-way set associative. This setup is 2.1% faster than the baseline, roughly equivalent to the standard virtual cache setup, showing that reduction of the TLB miss rate possible in the proposed protocol design is a significant contribution to the performance gain over the baseline. For the virtual cache setup, I evaluated a variant with the unified TLB size reduced to half as many entries (128 entries). This setup has equivalent overall performance (less than 0.1% difference) compared to the baseline. The ability to share TLB entries in this setup has offseted the loss from the reduced TLB size, reaffirming the effect observed in Section 3.3.1.

There are two noteworthy outliers in Figure 4.19. The `blackscholes` benchmark solves the Black–Scholes equation with 6 variables, over a time series of data. The data are allocated

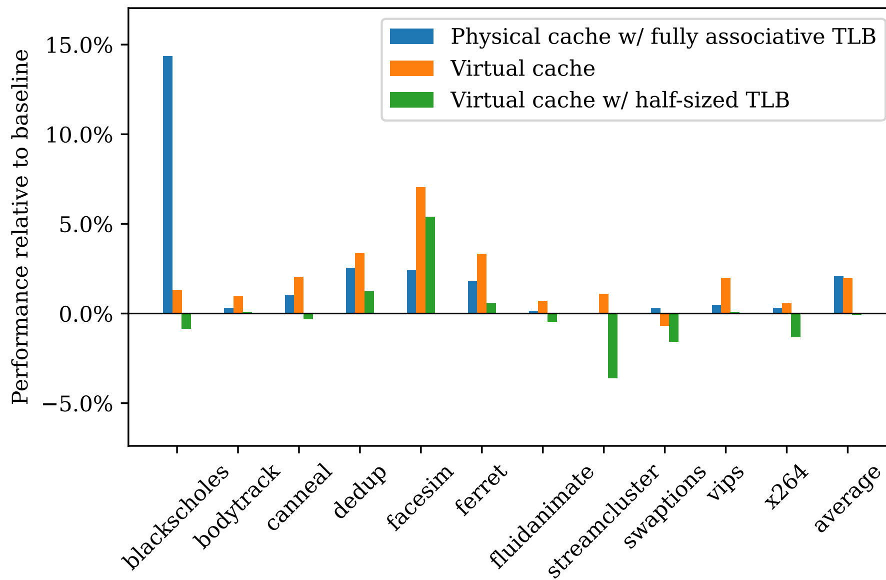


Figure 4.19 PARSEC benchmark performance comparison between physical, virtual cache and their variants

in 6 separate arrays and the way memory allocation is performed internally in the benchmark makes them highly aligned. Both virtual cache setups and the baseline setup are heavily disadvantaged by thrashing due to the associativity of 4 selected for their caches and TLBs. The physical cache with fully associative TLB is able to avoid one source of thrashing and therefore gains 14.4% over the baseline. The `streamcluster` benchmark has a trivial amount of sharing between threads while having a large working set [17], creating a higher level of load on the TLB and therefore disproportionately disadvantaging the half-sized TLB variant of the virtual cache setup.

For minibench, the TLB size and associativity is less relevant, and the performance depends on the protocol being used. For `scan` workload, none of the evaluated variants exhibit unique performance characteristics for small test size (4096 bytes). For larger sizes, 32768 and 524288 bytes, virtual cache has slightly worse performance as shown in Figure 4.20, likely caused by the L1/L2 delay attributed to the TLB placed in between them. If the TLB is stressed, which we deliberately for the size = 2097152, stride = 4096 configuration as shown in Figure 4.21, we see an opposite performance metric with virtual cache gaining over physical ones. For `modify` workload, for all 64-byte stride configurations, the performance favours physical cache to due high amount of L2 transactions (similar to Figure 4.20), and for 4096-byte stride the virtual cache is slightly better as the configuration is TLB-bound (similar to Figure 4.21).

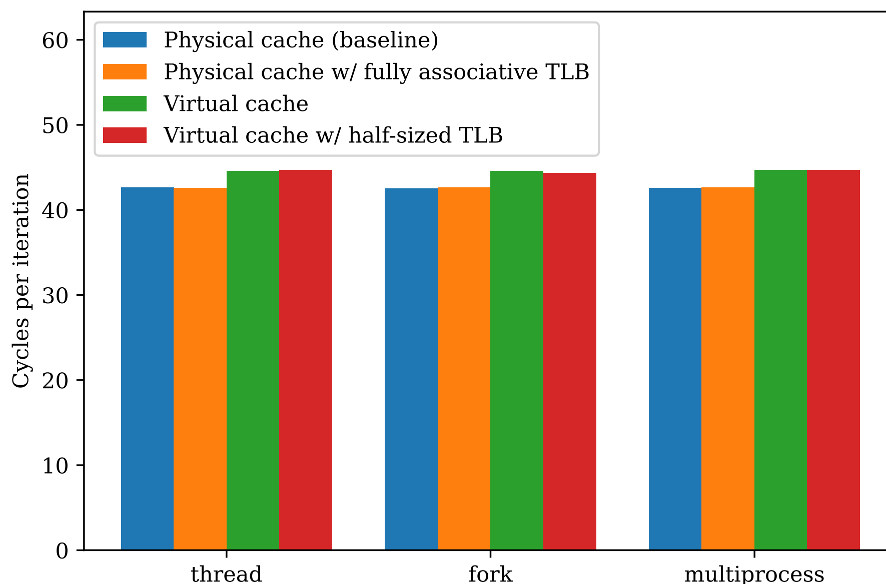


Figure 4.20 Summary of minibench evaluation, with a `scan` run, size of 32768 bytes, stride of 64 bytes

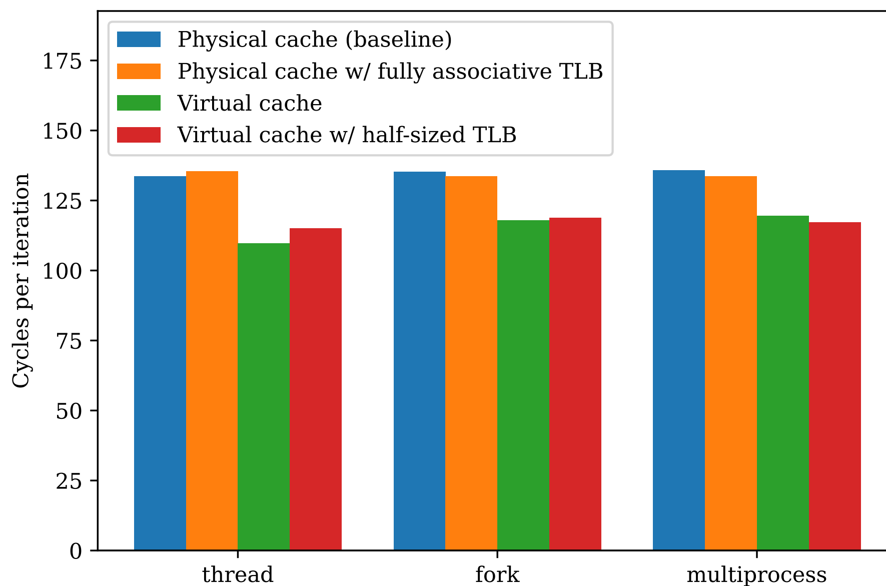


Figure 4.21 Summary of minibench evaluation, with a `scan` run, size of 2097152 bytes, stride of 4096 bytes

Overall, the evaluation of the proposed virtual cache coherence protocol demonstrates that it has competitive performance, meets the principles outlined in Section 4.1 and provides a good basis for a hardware implementation.

Chapter 5

Hardware Baseline and Virtual Cache Protocol Implementation

In addition to the simulation of the virtual cache coherence protocol, I concluded that a hardware implementation of it is desirable. Often there are edge cases that cannot be covered in the simulation process. Furthermore, an RTL implementation provides the opportunity for a more detailed evaluation. A real implementation running on an FPGA with an actual OS is also a good demonstration of the feasibility and practicality of the protocol.

5.1 Hardware Baseline

5.1.1 Processor Baseline

In order to pursue a hardware implementation of the virtual cache protocol, a processor baseline is needed. The processor needs to be suitable for general purpose computing, supporting multi-core processing and able to boot modern OS kernels such as Linux. Three open-source RISC-V processors available at the time of the implementation were evaluated, namely: Rocket, BOOM and Ariane.

Rocket [6] is an in-order scalar processor developed by UC Berkeley. It supports both RV64G and RV32G ISA, and later versions add support of the C-extension. Rocket is written in the Chisel hardware description language (HDL) [7], which is based on Scala. Rocket has a 6-stage pipeline, with two stages (PCGEN and IF) in the frontend and 4 stages (ID, EX, MEM, WB) in the backend. It uses TileLink [108] as its memory interface. Rocket comes with a complex data cache that is pipelined over four stages and has multiple miss status holding registers (MSHRs), which offer limited gains for an in-order core. Rocket is also used as the processor in v0.1 to v0.5 of lowRISC chip project [81].

BOOMv2 [26] is a superscalar OoO processor supporting RV64GC, also developed at UC Berkeley. A later version is named SonicBOOM [125]. Similar to Rocket, it is written in Chisel and uses TileLink as interconnect.

Ariane [124] (currently named CVA6) is an in-order scalar processor developed by ETH Zurich, implementing RV64GC. Interestingly, it utilises register renaming, scoreboarding and out-of-order writeback for issue and execution stages, which is more commonly found in OoO processors than in in-order cores. Such design techniques add complexity and increase the development and maintenance cost. Ariane by default uses AXI4 [5] as its memory protocol with signals extended to support atomic instructions (which is functionally equivalent to the AXI5, published at a later date). AXI4 however is not suitable for multi-core systems, so the multi-core variant of Ariane is based on the OpenPiton mesh network [9]. Ariane is used as the processor in v0.6 of lowRISC chip project.

A simple, straightforward, and easy-to-modify baseline is desired. BOOM, being a superscalar processor, is therefore excluded due to its complexity. Rocket's cache implementation is more complex than desired, and I would like to avoid Chisel; Chisel, being an embedded domain-specific languages (DSLs) inside Scala, is not only complex in its own but also inherits the complexity of Scala. Moreover, most electronic design automation (EDA) tools only support SystemVerilog and VHDL, so Chisel code needs to be compiled into plain Verilog before being input into the tools; this step is a lossy process, making debugging difficult. Ariane could serve as a good single-core baseline, but since OpenPiton is needed for multi-core systems, modification to the cache coherence protocol is harder and more constrained. Furthermore, Dr Jonathan Kimmitt discovered some I/O related issues in Ariane when implementing v0.6 of lowRISC chip project. I therefore concluded that neither Rocket, BOOM, nor Ariane, offered a suitable baseline for the implementation work.

SHAKTI-C [42] is another open-source processor. It is written in BlueSpec SystemVerilog [90] however, which was not open source at the time. Bluespec was only open-sourced in late 2020 [65]. For this reason, SHAKTI-C is not considered. BlackParrot [96] offers much but only became public at a later date, after the selection for the baseline.

I therefore decided to implement a brand-new processor to serve as the processor baseline in SystemVerilog. The implementation work started in 2019. In 2020, I started collaborating with Dr Daniel Bates and lowRISC CIC, and the project evolved into Muntjac. Muntjac is currently open-sourced in the Apache 2.0 license at lowRISC's GitHub repository [51].

5.1.2 Protocol Baseline

Starting a fresh processor design also gives me freedom to select any suitable interconnect protocol, not bound by the existing design of the selected hardware baseline. I designed

Muntjac to be modular, so both instruction and data caches communicate with the pipeline via generic valid-ready interfaces, allowing flexibility in cache designs and their protocol choice.

Despite this flexibility, I did still however need to choose a concrete interconnect protocol ahead of cache implementation because such protocols have a profound influence on cache designs. I selected a range of interconnect protocols as candidates; they include a custom 3 channel protocol and open standards such as AXI/ACE and TileLink.

These protocols make different design decisions on the number of channels, how they handle multi-beat transactions (bursts), and how message IDs are handled, and how they handle atomic memory accesses. I evaluated these protocols thoroughly and compared them in detail.

AXI

AXI [5] is short for Advanced eXtensible Interface, and is one protocol out of the Advanced Microcontroller Bus Architecture (AMBA) family. The specification is developed and published by ARM. The specification is royalty-free, but it does come with a restriction that if a design uses AXI and includes a CPU, then the CPU should either be licensed from ARM, or must not be compatible with the ARM ISA. Since Muntjac is not compatible with the ARM ISA, the license restriction would not apply.

The latest version of the AXI specification when the project began was AXI4, and the latest version of AXI as of the time of writing is AXI5. There is a trimmed down version called AXI-lite for peripheral IOs that have low performance requirements.

AXI/AXI-lite have 5 channels: Read Address (AR), Read Response (R), Write Address (AW), Write Data (W) and Write Response (B). Each channel has its own valid and ready signals and handshaking happens upon assertion of both valid and ready signals. AXI requires the valid signal be constantly asserted until the handshake happens. AXI requires that no combinational path exists between input and output signals.

AXI has two basic transactions:

- Read: host sends address, size and length on the AR channel, and device responds on the R channel with data and status.
- Write: host sends address, size and length on the AW channel, and sends data on the W channel. The device responds on B channel with status.

In AXI, all “size” signals indicate the size of a single burst, while “length” is the number of bursts minus 1. A “last” signal is required for channels that support multiple bursts (R/W

channels in AXI). The separation of size and length leads to a possibility called narrow bursts, when the size is smaller than the native size given the width of data signals but the length is non-zero. Narrow bursts are not supported by all AXI implementations.

AXI also places write address (AW) and write data (W) in separate channels. In AXI3, both the AW and the W channels carry transaction IDs, but it could be difficult to handle if addresses and data are not arriving in the same order. The ID of the W channel was subsequently removed in AXI4 and reordering of writes is forbidden.

AXI supports LR/SC by exclusive accesses. A read request can have `ARLOCK` set, and the device can respond with `EXOKAY` if it supports exclusive accesses. A subsequent write request can have `AWLOCK` set, and if the memory has not been modified between the read and the write request, the write will be accepted by the device and an `EXOKAY` status is returned. Otherwise, the memory is not updated and `OKAY` is returned. Starting in AXI5, the AW channel has an additional atomic operation signal `AWATOP`; the supported atomic operations are a super set of atomic operations supported by RISC-V.

ACE

ACE, short for AXI coherence extensions, is also from the AMBA protocol family. ACE extends AXI with 5 additional channels: Snoop Address (AC), Snoop Response (CR), Snoop Data (CD), Read Acknowledgement (RACK) and Write Acknowledgement (WACK). The AC, CR and CD are flow controlled using valid and ready signals like AXI channels, while the RACK and WACK channels are one-way and have no ready signals.

A typical read transaction that needs invalidation makes use of the AR, AC, CR, CD, R and RACK channels:

- The host issues a read transaction on the AR channel.
- The device forwards the address to the AC channel.
- Other hosts respond on the CR channel, optionally provide data on the CD channel.
- If data is present, the device forwards the data to the R channel; otherwise the device handles the read transaction itself, e.g. serve request from the cache or forward it to the next-level cache.
- The host completes the transaction by acknowledging on the RACK channel.

Write-back or eviction makes use of the AW, W, B and WACK channels:

- The host issues a write transaction on the AW channel.

- If the cache line is dirty, data is transferred on the W channel.
- The device responds on the B channel.
- The host completes the transaction on the WACK channel.

In ACE a cache line can be in one of 5 states: Invalid (I), UniqueDirty (UD), SharedDirty (SD), UniqueClean (UC), SharedClean (SC), i.e. it implements a MOESI protocol.

TileLink

TileLink [108] is a family of open-standard interconnect protocols originally designed by UC Berkeley Architecture Research group and subsequently SiFive. While ACE can support a MOESI cache coherence protocol, TileLink has a MESI-oriented design instead. There are two permission levels in TileLink: N (None), B (Branch) and T (Trunk), some quite atypical naming differing from normal `None, Read, Read/Write` nomenclature.

In TileLink, caches are viewed as a tree with a single device as the root (e.g. LLC), L1 caches as the leaf and intermediary caches (if any) as the middle nodes. For any particular cache line, all agents that contain cached copies of the cache line forms a subtree, known as *coherence tree* in TileLink. The point of write serialization is known as the `Tip`; a node on the path between the `Tip` and the root (including both) is called a `Trunk`, and children of the `Tip` are known as `Branches`. For example, when the L1 requests `Trunk` permission from the L2, the L2 requests `Trunk` permission from memory controller, then all agents have the cache line in the `Trunk` state. TileLink formally defines that only Trunk Tip with no Branches have write permission to the cache line to reflect the fact that only the L1 can write the cache line in this case.

There are 5 channels for a TileLink link: A, B, C, D and E. Out of these channels, A, C and E flow from hosts to devices, while B and D flow from devices to hosts.

- The A channel carries requests such as `Get` and `PutPartialData/PutFullData` messages for uncached memory accesses, or a `AcquireBlock/AcquirePerm` message for obtaining data and permission for a cache line.
- The B channel carries `ProbeBlock/ProbePerm` messages to invalidate cache lines from hosts.
- The C channel carries `Release/ReleaseData` voluntary cache line permission release (and writeback) messages, or `ProbeAck/ProbeAckData` messages in response to invalidation messages sent over the B channel.

- The D channel carries `AccessAck/AccessAckData` messages in response to `Get/PutPartialData/PutFullData` messages sent over A channel, or `Grant/GrantData` cache line permission grant messages in response to `AcquireBlock/AcquirePerm` messages sent over the A channel, or `ReleaseAck` messages in response to voluntary cache line release messages sent over the C channel.
- The E channel carries `GrantAck` message in response to `Grant/GrantData` messages sent over the D channel.

All channels are flow-controlled using valid and ready signals as is the case for AXI. TileLink, however, has a more relaxed combinational path rule, only requiring no combinational path from ready to valid but not in the other direction.

If sorted according to causal order:

- Uncached read: the host sends `Get` and the device responds with `AccessAckData`.
- Uncached write: the host sends `PutPartialData/PutFullData` and the device responds with `AccessAck`.
- Increase cache line permission: the host sends `AcquireBlock/AcquirePerm`, the device responds with `Grant/GrantData` and the host responds with `GrantAck`.
- Non-voluntary decrease cache line permission: the device sends `ProbeBlock` or `ProbePerm`, the host responds with `ProbeAck/ProbeAckData`.
- Voluntary decrease cache line permission: the host sends `Release/ReleaseData` and the device responds with `ReleaseAck`.

There are also atomic read-modify-write transactions and a hint/prefetch transaction. They are handled in a similar fashion as `Get` and `PutFullData`. Muntjac does not use these transactions so their details are omitted.

The channels are of ascending priority ($A < B < C < D < E$). Naturally, all responses are on a higher priority channel than the original request. If a link agent is waiting for the response on a channel, it must be able to process messages sent over higher-priority channels. For example, a host waiting for the response of `AcquireBlock` must be able to process a `ProbeBlock` message. However, a host waiting for the response of `ReleaseData` would not have to process `ProbeBlock` until it has received `ReleaseAck`. The priority requirement ensures that if a message is not processed, there must be a higher priority message in process. With limited (5) priority levels, this property guarantees the deadlock freedom of the TileLink network.

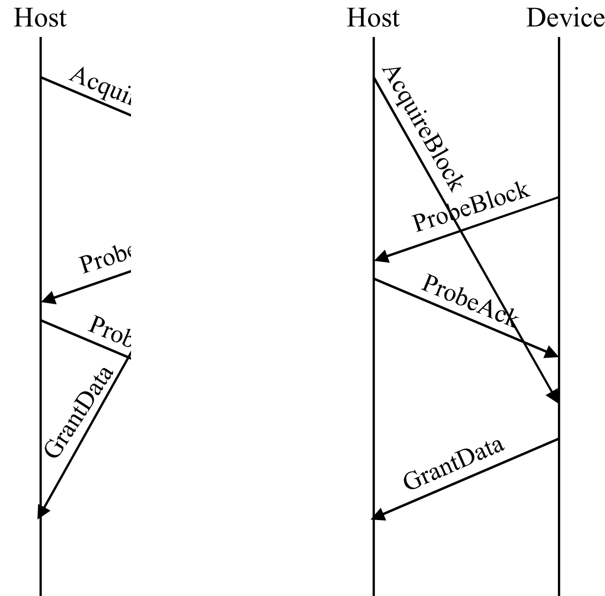


Figure 5.1 Ambiguous TileLink transaction if E channel is absent

TileLink does not guarantee any ordering of messages over channels. This relaxed requirement is the rationale behind the three-way handshake of cache line permission acquisition requests. If the E channel is absent, the transactions shown in Figure 5.1 are no different to the host; however in the left figure, the device would assume that the host no longer owns the cache line, while in the right figure, the device would assume that the host owns the cache line. This ambiguity is removed by requiring a `GrantAck` response to `Grant/GrantData` so the device would not send a `ProbeBlock` message until it has received the `GrantAck`.

There are three variants: TileLink Uncached Lightweight (TL-UL), TileLink Uncached Heavyweight (TL-UH) and TileLink Cached (TL-C). TL-C supports all messages and is the only variant of the family that supports cache coherency. TL-UH supports `Get` and `PutPartialData/PutFullData` (and hint/atomics) with their corresponding response messages. TL-UL supports `Get` and `PutPartialData/PutFullData` (without hint/atomics) and does not support multi-cycle bursty transfers. Since TL-UL and TL-UH do not support any messages on channel B, C and E, they only need channel A and D.

Custom protocol

I also considered a custom 3-channel protocol. It contains a `Req`, a `Resp` and a `Wb` channel. There are 6 types of messages, `Get/Upgrade`, `GetAck`, `Put`, `PutAck`, `Inv` and `InvAck`. This is essentially the physical address counterpart of the virtual protocol described in Section 4.3.

This is the simplest feasible protocol and was used during the early prototyping stages. BlackParrot also uses a similar 3-channel coherence protocol. However, during the implementation it soon became obvious that such a 3-channel protocol, as described, is flawed. The lack of acknowledge on `Put` means that a reordering of `Get` (on the `Req` channel) and `Put` (on `Wb` channel) is problematic. Consider the scenario where a host sends out a `Put` followed by a `Get` to the same address. Because the `Put` message does not require acknowledgement, if message on the `Wb` channel is delayed, the device may receive the `Get` first followed by `Put`, different from the sending order. The host would assume that it owns the block while the device would mistakenly believe that the block is released, causing inconsistency.

Industrial protocols have a larger number of channels to avoid this kind of problem. For example, TileLink requires acknowledgement on voluntary writeback (`Put`, or, in TileLink's term, `ReleaseData`) to ensure that different agents in the cache hierarchy have a consistent view of the cache line ownership. While fixing the issue without adding new channels is possible, e.g. adding reordering constraints across multiple channels, I conclude that additional complication would make a custom protocol not worthwhile given that TileLink only requires 2 additional channels and has no reordering constraints.

Comparison

One difference between AXI/ACE is the specification of transaction sizes. In AXI, two fields are used: size and length. TileLink instead has a single size field that can be only used to specify sizes of powers of two: if the size is smaller than the bus data width, it is a narrow transaction; otherwise the transaction is bursty. Data width conversion is complicated in AXI due to the possibility of narrow bursts; in comparison it is very simple for TileLink, and data width downsizing could even be stateless. TileLink's power-of-two requirement however means that burst length is also power of two, so DMA devices are harder to implement compared to AXI because they have to divide accesses to aligned chunks.

Another significant difference between AXI/ACE and TileLink is the usage of IDs. In AXI, the same ID can be used for multiple in-flight transactions. Devices are required to respond to transactions of the same ID in FIFO order and reordering is forbidden. In TileLink, the ID must not be reused until the transaction has been completed. AXI's design simplifies the design of simple hosts by allowing them to issue multiple transactions under the same ID and process them one by one; crossbar implementations are more complicated if two transactions to different devices are under the same ID, since the crossbar has to block the second request (otherwise the response to host could be out-of-order). On the other hand, TileLink crossbars only need to arbitrate messages going towards the same channel. Without reordering requirements, TileLink crossbars can be stateless.

In AXI, control and data signals are in separate channels; in TileLink they are sent in the same channel, and control signals are held constant for all data beats for bursty transactions. TileLink's approach makes interconnects and devices simpler because control and data will not be reordered; the fact that AXI4 dropped ID signals from the W channel manifests the disadvantage of separate control and data signals. A minor disadvantage of TileLink's design is that FIFOs either have to store redundant control signals or have to store control and data signals separately despite being from a single channel, but I consider it to be insignificant.

	AXI4-Lite	AXI5-Lite	AXI4	AXI5	ACE	TL-UL	TL-UH	TL-C
# of Channels	5				10	2		5
Data width	32 or 64	Up to 1024				Up to 4096		
Transaction Size	Full bus width	Up to bus width				Up to bus width	Up to 4096	
Burst Length	1		Up to 256			1	Length must be power of 2 Address must be aligned	
Control & Data	Separate					Combined		
"Last" Signal	Always true		Explicit			Always true	Implied	
Atomics	No		LRSC Only	Yes	Yes	No	Atomics Only	Yes
Coherence	No				MOESI	No		MESI
IDs	Optional	Required, same ID indicates FIFO				Required, same ID forbidden		

Table 5.1 Summary of interconnect protocol differences

Table 5.1 summarises differences between the interconnect protocols evaluated. AXI-Lite, TL-UL and TL-UH are mainly for device IOs and they are not suitable as the main interconnect protocol due to lack of atomic support needed to implement the entirety of RISC-V's A extension. I also ruled out the use of AXI4/AXI5. While AXI5 does support LR/SC and atomic operations, it is not cache coherent; the need of bus transactions for all atomic accesses is unfavourable. While traditionally it is believed that synchronisations are rare, the statement is no longer true given the rise of multi-threaded programs. To avoid data races, very fine-grained locking is becoming widely adopted, and their performance and overhead has been greatly improved by ParkingLot [98]. The Rust programming language [114] takes further steps to eliminate data races by ensuring all data shared between threads are protected by thread-safe reference counters and locks, all of which use atomic operations underneath. As a result of the fine granularity, the vast majority of atomic memory accesses are uncontended. The performance in such cases becomes critical, and the requirement of bus transactions for all atomic operations is undesirable.

Considering simplicity and the number of channels, I ultimately selected TL-C over ACE for cache coherent links, TL-UH for uncached accesses and TL-UL for I/O memory access. The TileLink specification includes provision for update-based protocols; Muntjac does

not implement them. Removing the need to support update-based protocols saves logic by eliminating the data signals and the possibility of having bursty messages on the `B` channel.

One additional aspect worth mentioning is not about the intrinsic design of protocols, but their current adoptions. Many existing third-party IPs use AXI. So for compatibility/interoperability reasons, a bridge needs to be feasible between the protocol chosen (TileLink) and AXI. Fortunately, TileLink to AXI bridge is fairly trivial; AXI to TileLink bridge requires splitting AXI transactions into chunks due to the address alignment requirements of TileLink, but I consider this burden to be acceptable. I have implemented these bridges to be used as part of Muntjac component collections.

5.2 Muntjac – Open Source Collections of Processor and

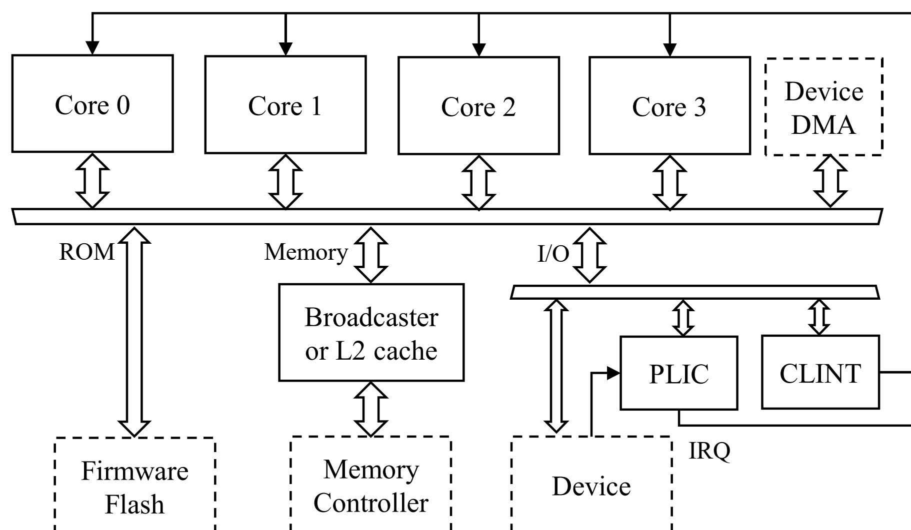


Figure 5.2 Example multi-core Muntjac system. All solid-border components are provided by Muntjac.

As part of Muntjac, I have implemented a collection of open-source components that can be used to build a multi-core, Linux-capable system-on-chip. Figure 5.2 contains an example multi-core system, with all solid-border components provided. This includes a 64-bit RISC-V core, a cache subsystem, and TileLink interconnect supporting cache-coherent multi-core configurations and I/O. Each component is easy to understand, verify, and extend, with most being configurable enough to be useful across a wide range of applications. The focus of Muntjac is on having clean, well-tested designs with clear routes to further customisation and improvements, with performance competitive with other open-source projects.

Muntjac also have a strong focus of verification, taking guidance from Ibex [82]. Work on testing and verification was undertaken in collaboration with Dr Daniel Bates.

The motivation and rationale for building Muntjac to begin with is to be a test bed for computer architecture research. Systems that contain I/O devices, receive various interrupts and run an OS kernel or larger programs can be prohibitively slow in simulation (even with binary translations) and therefore simulators would always have to trade fidelity for speed. Muntjac aims to provide a solid baseline for experimentation, removing the need to build and test huge swathes of general-purpose infrastructure needed to support a single novel component. Apart from research purposes, however, it is hoped that Muntjac can be valuable in many other situations:

- Education: the complete code for a high-quality microprocessor is available, with documentation and examples, and lends itself to self-contained extensions (e.g. a new branch predictor). I documented some of the nuances of real processor design, and some of the key design decisions made during the process.

- Industry: many real-world applications value "time to solution" over absolute performance or energy efficiency. Muntjac is a reliable, configurable, complete starting point, allowing engineers to focus on the unique selling points of their target system.

5.2.1 Core Overview

The Muntjac core is scalar, in-order, and supports the RV64GC instruction set with machine and supervisor ISA. Floating-point extensions (F and D) are optional and can be configured with SystemVerilog parameters. The list of supported instruction-set extensions and the standards they conform to is detailed in Table 5.2.

The Muntjac core is designed with modularity in mind. It aims to be easy to understand, verify and extend. As shown in Figure 5.3, the core is separated into 4 components. The frontend and the backend are loosely coupled and valid-ready signals are used for stalling and back-pressure. The instruction and data caches are also loosely coupled with the frontend and backend, respectively. Separating pipeline design and cache design allows a differently designed cache to be swapped in easily without having to adjust the design of the pipeline.

In general, we choose distributed stall signals over a global stall signal or a stall-free design. Valid-ready signals are widely used both within components and between components, so each component and each pipeline stage can be mostly self-contained. Skid buffers are used when distributed stall signals start to cause timing issues.

Standard	Version
RV64I: Base Integer Instruction Set, 64-bit	2.1
M: Standard Extension for Integer Multiplication and Division	2.0
A: Standard Extension for Atomic Instructions	2.1
C: Standard Extension for Compressed Instructions	2.0
F: Standard Extension for Single-Precision Floating-Point	2.2 (Optional)
D: Standard Extension for Double-Precision Floating-Point	2.2 (Optional)
ZiCSR: Control and Status Register (CSR)	2.0
Zifencei: Instruction-Fetch Fence	2.0
Machine ISA	1.11
Supervisor ISA	1.11

Table 5.2 List of RISC-V instruction-set extension standards implemented

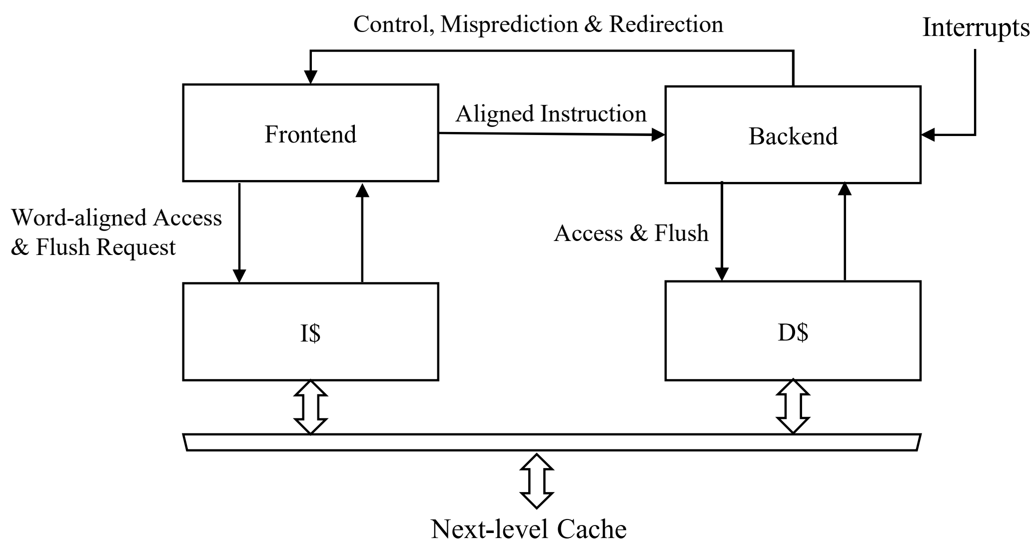


Figure 5.3 High-level overview of Muntjac core components

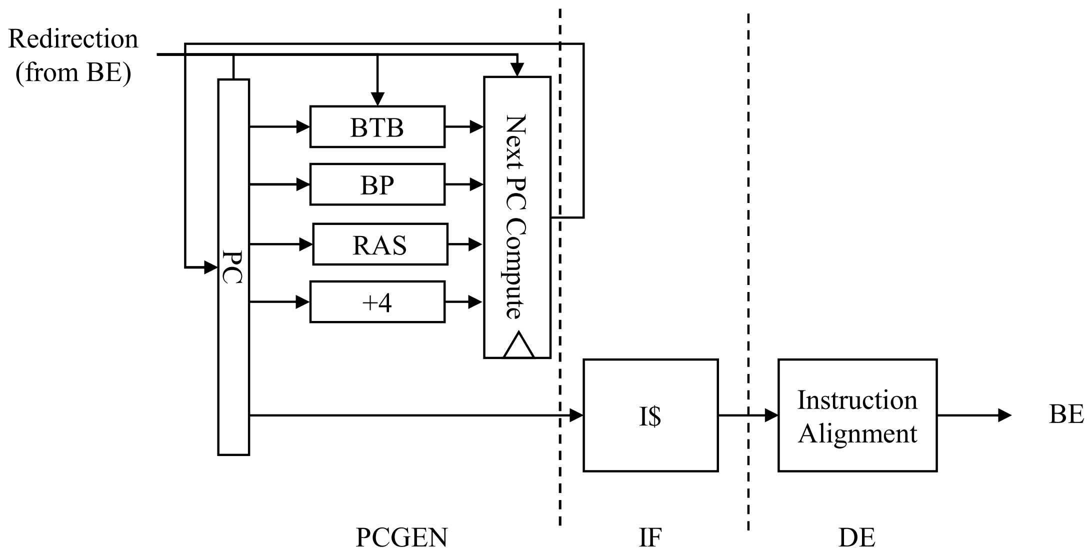


Figure 5.4 Muntjac frontend design

Muntjac's frontend design is illustrated in Figure 5.4. It is a fairly classical frontend design, except that variable-length instructions are supported. There is a separate PCGEN stage that generates the next PC, without peeking into the fetched instruction bytes.

Muntjac supports the compressed instruction (C) extension. The C extension is desirable because it brings significant code size reductions, improves instruction cache efficiency and is also required by most Linux distributions. I made a deliberate choice to not allow the C extension to be turned off via a Verilog parameter, because doing so requires us to add support for the misaligned instruction exception. A RISC-V misaligned instruction exception is triggered at the jump instruction that causes it; this is the only possible exception generated by jump instructions, so requiring the C extension can make the backend simpler.

It should be noted that the C-extension does not only adds 2-byte compressed instructions, but also relaxes the alignment requirement of 4-byte instructions to be only 2-byte aligned. The relaxed alignment requirement does mean that Muntjac has to deal with the complexity of a misaligned 4-byte instruction, potentially crossing a cache line or even a page boundary. For any instructions that are not 4-byte aligned, two bytes will be fetched first; the following bytes might be prefetched, but the exception triggered during the prefetch would be discarded if the first two bytes indicate that the instruction is compressed.

The complexity of compressed instructions and misaligned instructions is entirely handled within the frontend; the instruction cache only needs to support accesses aligned to word

(32-bit) boundaries. When the PCGEN stage generates a PC, it will also generate a mask indicating which half-words of the fetched word are significant. For example, when a branch lands on a misaligned location (i.e. $PC\%4 = 2$), a mask of 0b10 is generated; when the PCGEN stage predicts that control flow will deviate after an instruction that ends on a misaligned location (e.g. a misaligned 4-byte predicted-taken branch), a mask of 0b01 is generated; a mask of 0b11 is generated for most other scenarios where all half-words are significant. The instruction alignment logic takes the fetched words and the masks and re-segments into individual instructions to be decompressed and decoded.

The branch target buffer (BTB), return address stack (RAS) and branch predictor are used to predict the next PC. In the current version of Muntjac, a direct-mapped BTB is used. The branch target buffer will output 3 pieces of information:

- the type of instruction: whether the instruction is a conditional branch, jump, call, or return instruction, or not a control flow instruction at all;
- the target PC in case the instruction is a branch or a jump;
- whether the second half-word is part of the instruction stream (in case the control-flow instruction ends on a misaligned location).

The RAS is pushed when a BTB predicts a call or a yield instruction, and is popped when BTB predicts a return or a yield instruction. The RAS is implemented like a ring; pushing when the RAS is full will overwrite the first entry, and popping when the RAS is empty reads the last entry. A separate set of buffer pointers are maintained that are only adjusted when a call/return/yield instruction is committed; in case of a misprediction, this counter will override the speculative counters to re-balance the return address stack, so that a single misprediction will not cause cascade RAS mispredictions.

The branch predictor is used when the BTB predicts a conditional branch. Currently, the branch predictor implemented is a simple bi-modal 2-bit saturating counter, but like other components, the interface between frontend and branch predictor is well-defined and a different implementation can be easily swapped in.

The backend notifies the frontend for all control-flow instructions committed whether correctly predicted or not. The frontend makes use of this information to train the branch predictor, and in case of a misprediction, also trains the BTB, adjusts RAS pointers, and re-initiates the fetch.

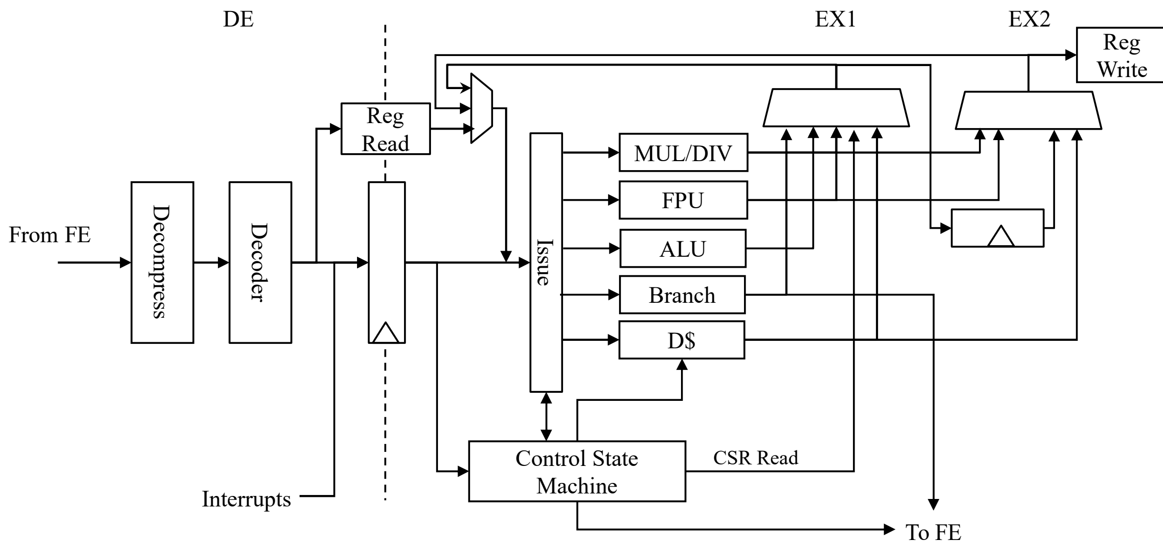


Figure 5.5 Muntjac backend design

Backend

The aligned instructions from the frontend are expanded into 4-byte instructions if they are compressed, decoded and have operands fetched in the DE stage as shown in Figure 5.5. The issue logic manages control, data and structural hazards, and issues instructions into one of the functional units.

The arithmetic logic unit (ALU) and the branching unit complete in a single cycle, while the latency of other functional units may be a variable number of cycles. To hide the latency of data cache accesses (usually 2 cycles for a cache hit), there are two execution stages, EX1 and EX2. All instructions progress from EX1 to EX2 before their result is written back; a register is placed between the two stages to hold the result if the functional unit completes before it could progress into EX2.

CSR accesses and other system instruction executions are handled outside the normal data flow. When a system instruction is decoded, the issue logic blocks it from continuing down the pipeline, and instead waits for all previously issued instructions to commit (or trap). It will then dispatch the instruction to the control state machine when the pipeline is empty. After the control state machine deals with the instruction, it will inject the execution result (e.g. CSR value read) into the pipeline. This design ensures that all backend control signals are stable, so functional units do not have to latch control signals internally; global states can only change when the pipeline is empty.

Except for the data cache interface, no functional units nor the control state machine may generate exceptions. Insufficient privilege or illegal CSR accesses are all decoded as illegal instructions by the decoder. If a load or store instruction triggers an exception, all instructions

in flight are cancelled by setting their destination register to x0. This simple cancellation process is made possible by the fact that all functional units other than the data cache are stateless and cannot generate exceptions, and that the control state machine is outside the

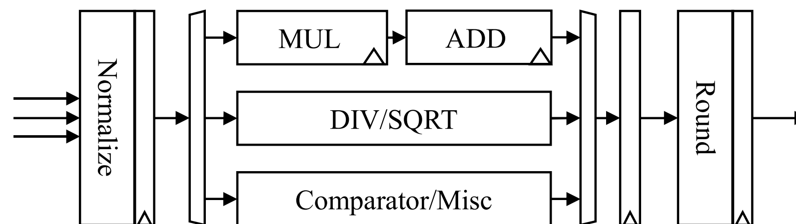


Figure 5.6 Muntjac FPU design

An FPU is available with Muntjac, supporting both single-precision and double-precision floating-point arithmetics. It is divided into 3 stages as shown in Figure 5.6. The “normalize” stage converts subnormal numbers into normalized form and decodes zero, infinity and NaNs. Single-precision floating-point numbers will be expanded to double-precision at this stage so arithmetic pipelines are unaware of the difference. There is a multiplication-and-add pipeline, which as its name implies, handles multiplication, add, and fused multiplication-and-add (FMA). Division and square root using long division are in the same pipeline. Another pipeline handles comparison, min-max and other miscellaneous operations. After the arithmetic pipeline, the result is fed into one of the single-precision, double-precision or integer rounding modules to convert from the internal normalized form back to IEEE format or to integers. The number of cycles depends on the operation and ranges between 3 and 59. Unlike many other implementations, we opted not to use a recoded format internally, so that NaN-boxing behaviour is more easily implemented and hard-to-test bugs related to recoding can be avoided.

I provide a few options for floating-point support:

- No FP support: No FP registers are supported, and FPU is not instantiated

- Full FP support: FP registers are supported, and FPU is instantiated
- FP register only: FP registers are supported, but FPU is not instantiated. In this mode, FP load and store operations are supported and handled by the hardware, but other FP operations will cause an illegal instruction exception. This mode can be used if the user only uses FP instructions occasionally on cold paths, wants the core to appear as RV64GC, but does not want to bear the cost of FPU. In this mode, M-mode firmware can trap and emulate the floating-point operations, but trapping is not needed when the OS merely saves or restores FP contexts without performing actual computations.

The FPU is loosely coupled as other components and communicate with the pipeline using valid/ready interfaces. This means that it is possible to have multiple Muntjac cores sharing a single FP unit (this approach was demonstrated in AMD's Bulldozer processor where a pair of cores shared a single FPU [85]), but this is not yet implemented.

5.2.3 Cache Design

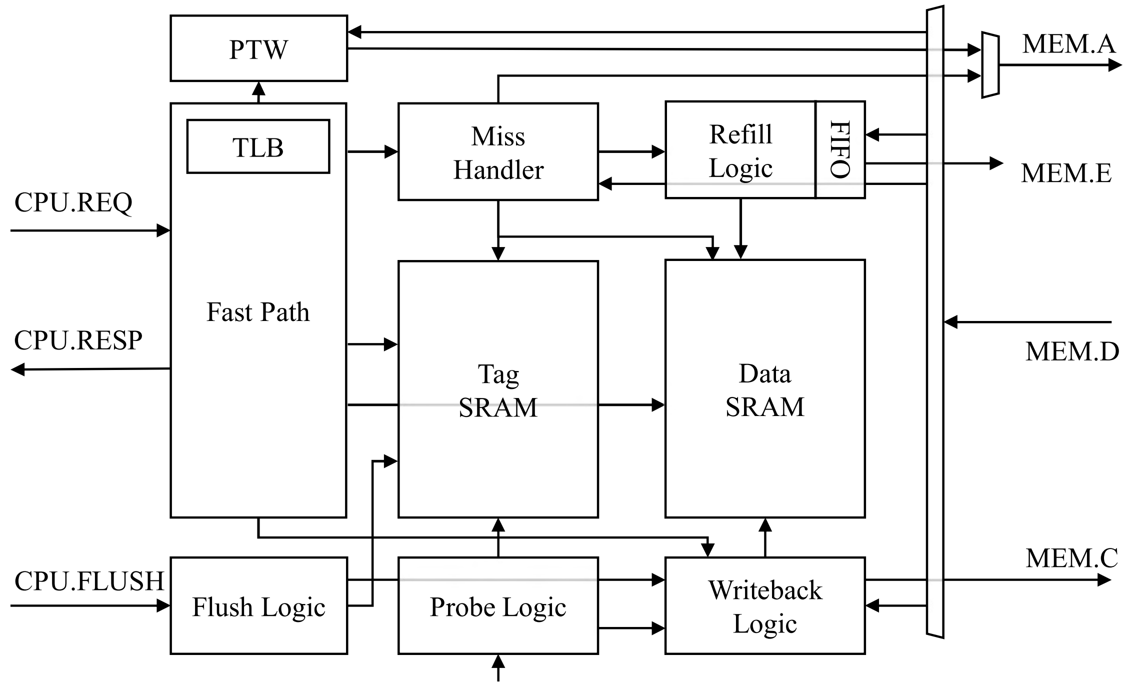
When a cache miss happens, caches would need to refill the relevant cache line from the main memory or the next-level cache; the access would be replayed afterwards. There are two common approaches in processor design on how to deal with the replay of requests: it can either be managed by the pipeline or be dealt with within the cache. Muntjac prefers loosely coupled interface, so both instruction and data caches communicate with the pipeline via generic valid-ready interfaces, and the replay for access requests are handled internally in the caches without the help from the frontend or the backend.

Muntjac provides cache implementations using the generic interfaces that comply with the TileLink protocol.

Data Cache

Muntjac's data cache implements the TileLink Cached (TL-C) protocol for its memory-facing interface and is therefore multi-core-capable. It is organised into several state machines as shown in Figure 5.7. The cache is set-associative, and for each way, tags and data are stored in separate single-port SRAMs. My initial implementation used simple dual-port (1-read 1-write) SRAMs, but I switched them to single-port SRAMs to minimise transistor usage as benchmarks showed only a small performance difference. Data SRAMs are 64-bit wide and do not have byte masks for writes.

When an access request arrives from the backend (via `CPU.REQ`), a parallel lookup to all of the tag SRAM, the data SRAM and the TLB take place as shown in Figure 5.8. The



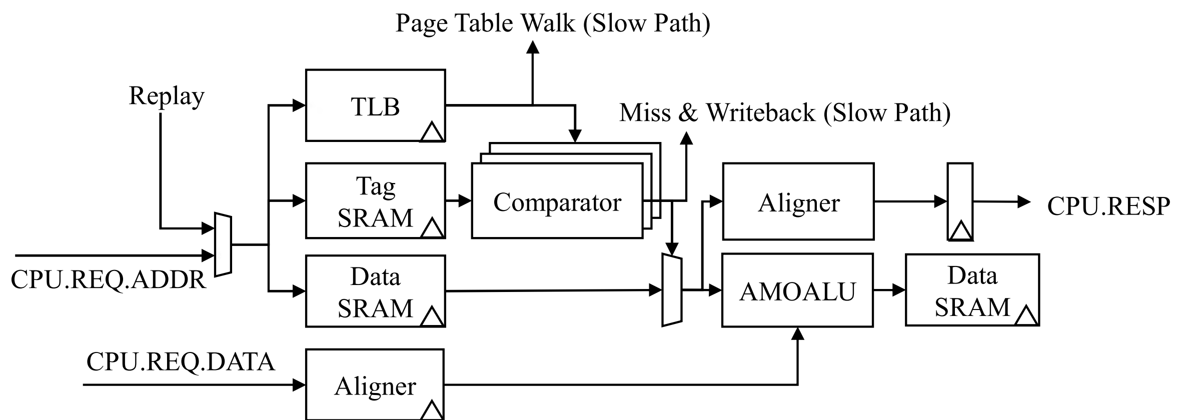


Figure 5.8 Muntjac L1 data cache fast path

physical address translated by the TLB is used for tag comparison (i.e. this is a PIPT cache). For a load or atomic fetch-and-update request, if everything hits, the data word fetched and multiplexed will be aligned if the access is narrow (less than 64-bit) and returned to the pipeline.

The cache has its own ALU, the atomic memory operation ALU (AMOALU), for atomic operations and narrow stores. For a store or an atomic fetch-and-update request, the supplied data from the pipeline will first be aligned, i.e. have the bytes shifted to their position in the containing 64-bit word for a narrow store. The current value from the data SRAM and the aligned data will then enter AMOALU, recombined with the current value for the inactive bytes before being written back into the data SRAM. For example, say we have a 64-bit value `0x12345678_DEADBEEF` at address `0x1000`, and we want to perform a 32-bit atomic add operation to add value 1 to address `0x1004`. The AMOALU will

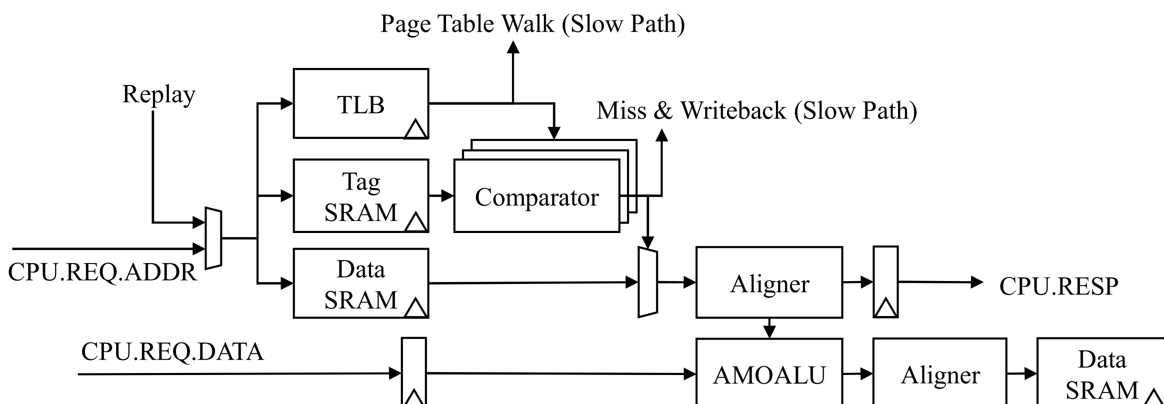


Figure 5.9 Alternative Muntjac L1 data cache fast path design

An alternative design is to not perform byte recombination in the AMOALU, as shown in Figure 5.9. Take the same example as the last paragraph, the raw value from data SRAM needs to be preprocessed to `0x12345678` (MSBs ignored). AMOALU receives this value, adds `1` and output `0x12345679`. The value is then broadcasted to all 64-bits as `0x12345679_12345679`, and a byte mask is used when writing back to the data SRAM so that only the MSBs are overwritten. This approach makes the AMOALU slightly simpler; however, the critical path is considerably longer. Using byte masks can also make future

implementation of ECC harder as checksums need to be computed from all bits. Therefore, recombining bytes is preferred to using byte masks on SRAMs.

The SRAM accesses are arbitrated between all subcomponents of the cache, so the SRAM lookups might be blocked when another subcomponent (e.g. the probe logic) is accessing it, in which case the request will be replayed. Similarly, a TLB miss will trigger a page table walk and a TLB refill, and the request will be replayed afterwards. If the accessed cache line is not present in the cache, the fast path will trigger the miss handler, and optionally trigger the writeback logic if eviction is needed for a replacement.

The fast path only interacts with the SRAMs and all TileLink protocol details are handled in other subcomponents. To avoid SRAM access races, refill, probe, flush and writeback logic are mutually exclusive and at most one of them can be active at a time. Upon a cache miss, the miss handler will send a TileLink A-channel `AcquireBlock` message. The corresponding D-channel `Grant`/`GrantData` response is processed by the refill logic, which updates the tag and data SRAM and sends back `GrantAck` message over channel E. Because writeback logic (C-channel) might be active when the response is received and TileLink mandates D-channel messages takes priority over the C-channel, a FIFO is needed for the refill logic. I/O accesses are not specially treated by the fast path; they are treated as cache misses and the miss handler will still send out an `AcquireBlock` message. If the address space accessed is I/O, the request will be responded with a denied `Grant` message, and then the miss handler will retry the request using `Get` or `PutPartialData` uncached access.

Writeback logic is responsible for both voluntary releases (e.g. eviction or a flush) or for responding to probes initiated by the next-level cache. In the case of a voluntary release, writeback will issue `Release`/`ReleaseData` and wait for `ReleaseAck`. Probe logic is activated when a `ProbeBlock` message is received over the TileLink B-channel, and it will look up the tag and leave the task of sending `ProbeAck` or `ProbeAckData` message to the writeback logic. The probe logic is required even in a single core configuration, e.g. to keep instruction cache and page table walkers coherent.

The RISC-V instruction set includes LR/SC instructions, which may cause livelock in a multi-core setup; a forward progress requirement is thus included in the specification. In Muntjac, when a cache access misses and the cache line is refilled, a 16-cycle timer starts ticking. Within 16 cycles, any `ProbeBlock` request to the cache line is blocked. This invalidation protection is lifted when a memory operation to an address other than the load-reserved address has been made, another cache miss happens or when the timer expires, whichever happens first.

Muntjac's data cache, as it is currently implemented, performs no reordering of operations and treats all fences as no-ops, so it is compatible with a sequentially consistent memory

model. This is means that Muntjac is currently compliant with both RISC-V weak memory model (RVWMO) and RISC-V total store order (RVTSO). However, it should be noted that the cache is implemented in this way solely due to simplicity rather than a deliberate design decision. Future changes and improvements could weaken it so that it is no longer compliant with RVTSO. RVWMO compliance is however guaranteed as mandated by relevant RISC-V standards.

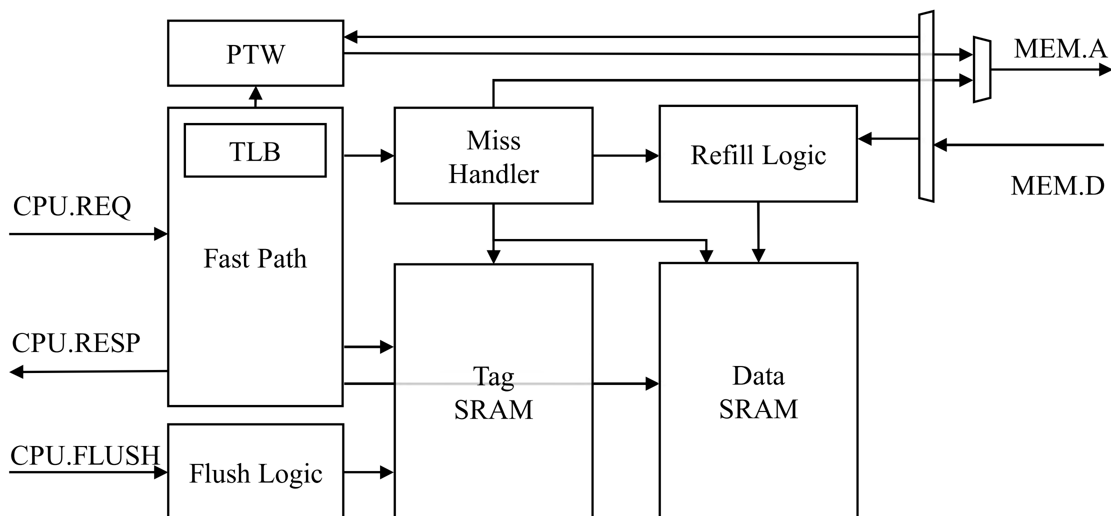


Figure 5.10 Muntjac L1 instruction cache subcomponents

Muntjac's instruction cache is derived from the data cache but with unnecessary features removed. The high-level structure is portrayed in Figure 5.10. As a derivative, the overall design and operation are similar to those of data cache.

In RISC-V, an explicit FENCE.I instruction is necessary to ensure modified code is visible to instruction fetch. Therefore, I implemented a non-coherent version of the instruction cache; it will be fully flushed when a FENCE.I or SFENCE.VMA instruction is executed. All cache coherency related logic is removed from this instruction cache, and the TileLink variant it conforms to is TileLink Uncached Heavyweight (TL-UH). When the cache misses, a bursty `Get` request is sent instead of `AcquireBlock`. A failed `Get` request will cause an instruction access fault immediately rather than initiating a retry as an I/O memory access. If needed, a coherent instruction cache that communicates via TL-C is also available.

Write support is not needed in the instruction cache which simplifies the fast path compared to the data cache, shown in Figure 5.11. Also, unlike the data cache which needs to support variable size accesses, the instruction cache supports only 32-bit access;

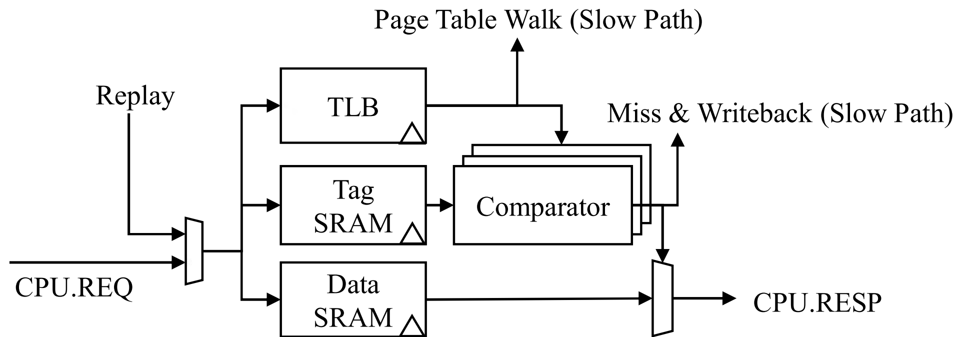


Figure 5.11 Muntjac L1 instruction cache fast path

the alignment stage is therefore also removed from the instruction cache (note that the compressed instruction handling is performed entirely in the frontend). This narrower access also allows us to reduce the data SRAM's width to 32-bits.

SRAM Word Interleaving

The native word width of the data cache's data SRAM is 64-bit, and 32-bit for the instruction cache's data SRAM. However, the bus width is often configured to be wider than 64-bit to improve bandwidth and reduce latency caused by a high number of beats in a bursty transaction. Adding a TileLink data width converter could solve the data width discrepancy, but the performance is not optimal.

In Muntjac's set associative cache design, data SRAMs are already banked; each way needs its own data SRAM bank to allow parallel lookup for cache accesses and independent updates. For example, a 64-bit 4-way data cache therefore has a SRAM bandwidth of 256 bits/cycle for lookup. An interleaving technique would allow all the bandwidth to be utilised for refilling as well.

A naive design would have each data SRAM bank storing data for a specific way, and data in the same cache line would be indexed by their offsets within the cache line. With interleaving, words in the same cache line might not be stored in the same bank of data SRAM. The index and way number used to access a word depends on both its offset within the cache line and the way number of the cache line.

Figure 5.12 and Figure 5.13 illustrate the interleaving mechanism and how data access is performed with this scheme with a hypothetical 4-way cache where each cache line contains 4 words. ABCD indicates the way number of a cache line, and 0123 indicates the offset within the cache line. A0-3 represents words in a single cache line, but they are spread out over different banks.

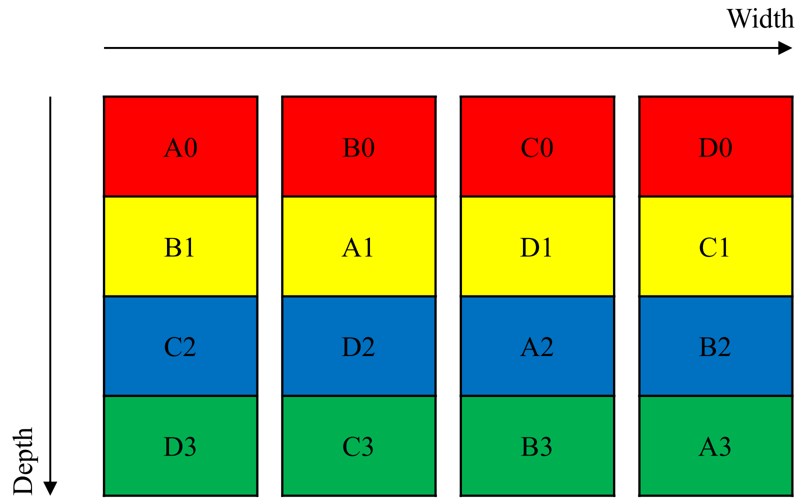


Figure 5.12 Parallel data SRAM lookup across multiple ways, given a fixed offset

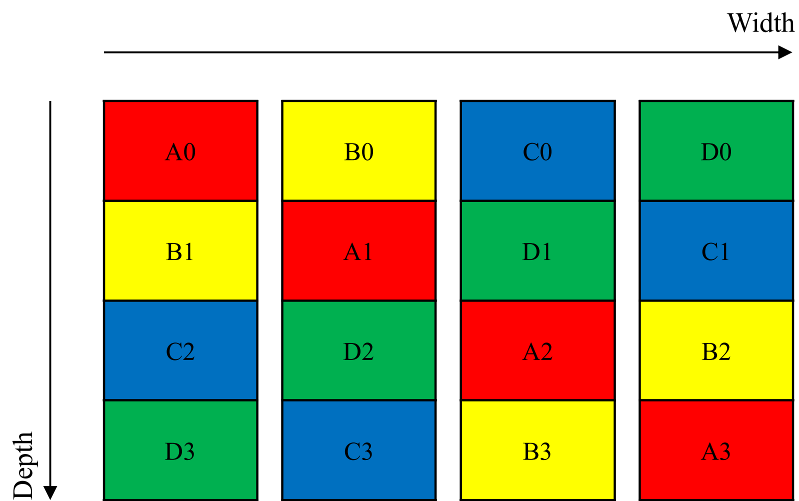


Figure 5.13 Wide data SRAM access of multiple words within the same cache line

Cells with the same colour are simultaneously accessed in the same cycle. Figure 5.12 demonstrates a fast-path read operation that must simultaneously read all ways of a particular set in parallel. The same index is used for SRAMs in each way, so words of the same offset are fetched for all ways (all cells sharing the same colour have the same offset). Figure 5.13 on the other hand reflects the scenario where wide data access is needed for writeback or refilling. A different index is used for each SRAM, so that all fetched words belong to the same cache line (all cells sharing the same colour are in the same way).

This interleaving scheme reduces the number of cycles needed for writeback and refilling, expanding the maximum bus width from 64 bits to 256 bits for a 4-way set-associative data cache without the need to expand the bit width of each SRAM or adding TileLink adapters. Similarly, a 4-way set associative instruction cache supports a maximum bus width of 128 bits.

5.2.4 Multi-core Support

Both data cache and instruction cache have a memory-facing TileLink link, and they are aggregated as shown in Figure 5.3. The aggregation is performed by a passive stateless 2:1 multiplexer that only switches messages based on source IDs. Currently 4 IDs are used per core: the data cache and instruction cache are allocated with one ID each, and their page table walkers are each allocated with a distinct ID.

The instruction cache and data cache are considered as distinct hosts in the TileLink network, because they are not inherently coherent to each other; when an instruction cache requests a dirty cache line that resides in the data cache of the same core, the dirty data should be written back and used, instead of the stale data in the next-level cache or the main memory. Similarly, the page table walker for the data TLB is considered a distinct host from the data cache itself, so that up-to-date PTEs are used.

These requirements means that each core consists of multiple hosts from the TileLink network's perspective. Therefore, there are no fundamental differences between a single-core Muntjac system and a multi-core Muntjac system. Coherency must be maintained by a simple probe broadcaster unit or a directory at shared L2 cache.

In a typical system like Figure 5.2, TileLink links from each core and possibly DMA-capable devices will be further aggregated with a stateless m:1 multiplexer; the link will then be split into multiple links with a stateless 1:n demultiplexer depending on address ranges and sink IDs. The handling will differ depending on the properties of the address spaces:

- IO: all caching `AcquireBlock` requests are denied. `Get`/`PutPartialData` are allowed and further demultiplexed to core local interrupt (CLINT) controller, platform-level

interrupt controller (PLIC) or devices. Interrupt controllers and devices follow TileLink Uncached Lightweight (TL-UL), which does not support bursty transactions.

- ROM: `AcquireBlock` messages that request `Trunk` (read-write) permission are denied. `PutPartialData` requests are denied. `Get` is allowed, and read-only `AcquireBlock` requests are converted into `Get` requests by a "TL-C ROM terminator" component.

- Memory: unlike IO and ROM, memory-like addresses can both be cached and modified. A broadcaster or an L2 cache is required as previously mentioned. Ultimately, after

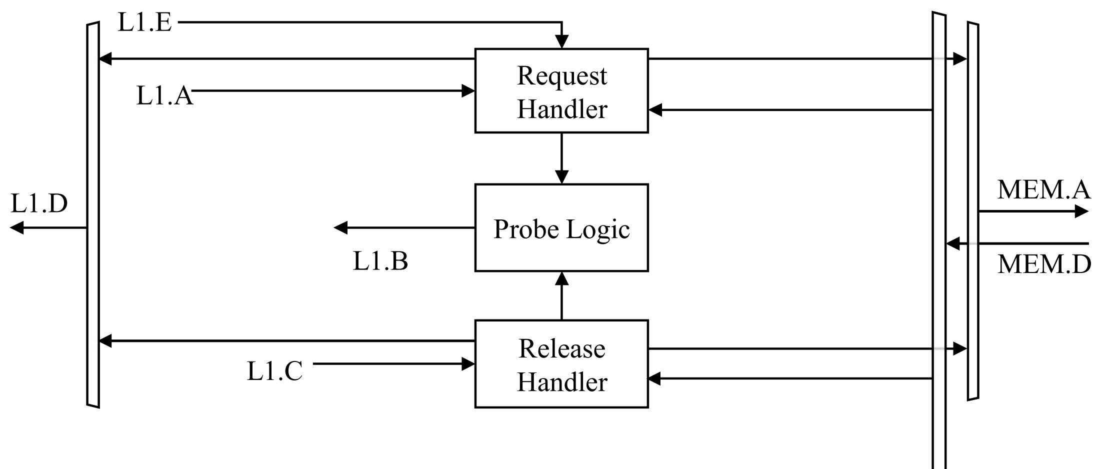


Figure 5.14 Simple broadcasting bus to bridge TL-C multiple hosts to a TL-UH port

For 1 or 2 cores, a broadcaster like Figure 5.14 is simplest and requires no additional SRAMs. For each incoming request on the L1's A channel, the broadcaster will send out a `ProbeBlock` to all caching hosts except the initiator. The message will be converted to `Get` only after all `ProbeAck` messages are received. `ProbeAckData` and `ReleaseData` are converted to `PutFullData` while `Release` is responded by the broadcaster directly with `ReleaseAck`.

A broadcaster will generate probing traffic to all cores for all requests even if these cores are not using the cache line. This will hurt the performance of L1 caches. For multi-core systems, a directory-based protocol should be used and an implementation of such L2 is available.

L2 Cache

Muntjac provides a reference L2 cache design. This cache uses TL-C for both the CPU-facing link and memory-facing link, so despite its name, it could be cascaded and used as L3 caches as well. This cache design can also be used as individual banks of a larger cache, multiplexed by address.

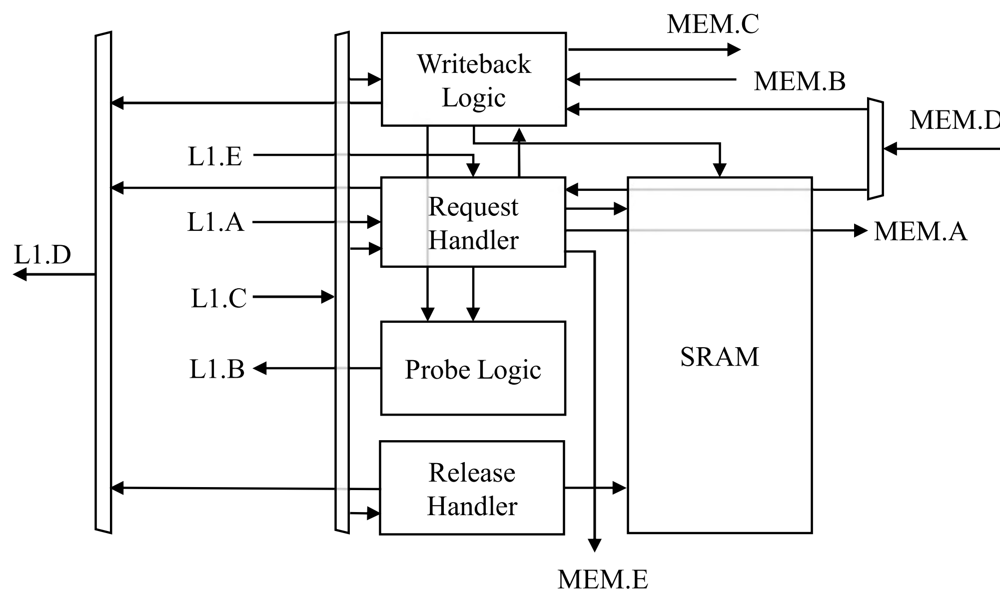


Figure 5.15 Muntjac L2 cache design

The cache consists of SRAMs for tags and data, probe logic, writeback logic and handlers for requests and releases, as shown in Figure 5.15. Multiple copies of request and release handlers can exist for parallelism, allowing messages for different addresses to be handled differently. Typically the number for each type of handler matches the number of cores. It should be noted that to ensure correctness, only one of the subcomponent as shown in Figure 5.15 can be active for a particular address.

The probe logic is a very simple sequencer. A cache line may be shared by multiple L1 caches, so invalidation could potentially be multicast. TileLink is a unicasting protocol, so a sequencer is needed to separate the multicasting invalidation to a number of unicasting `ProbeBlock` messages.

The request handler, as its name suggests, handles request messages like `Get` or `AcquireBlock` from A channel. If cache access hits and no probing is necessary, the

response gets sent back on the D channel. For `Grant/GrantData` responses, recipient `GrantAck` on E channel will bring the request handler to idle state for new requests. The request handler is also responsible for refilling the cache if the cache access misses. To reduce latency, the refilled data is simultaneously forwarded back to the L1 and written back to the data SRAM. If cache line eviction is necessary before refilling, probing of the evicted cache line and dirty data writeback is delegated to the writeback logic.

When probing is necessary, the request handler sends out the probe via probe logic, and is expected to handle the `ProbeAck/ProbeAckData` itself, as well as `Release/ReleaseData` messages to the same address. This requirement is essential for correct operation of TileLink. If this is not the case, when another core issues a `Release` to the same cache line while the request handler is waiting for its probe responses, a release handler will be allocated to operate on the same cache line, causing an address conflict and a potential data corruption. Whenever a C channel message with data is received, the request handler simultaneously writes it back to the SRAM and forwards it to the requester of the original initiating A channel message. The handler will read data from SRAM and respond to the requester if none of the C channel messages contain data.

The writeback logic is responsible for both eviction and probes from next-level caches. Similar to the request handler, it sends out the probe via probe logic and handles all C channel messages to the cache line of concern. Though, in this case, data are forwarded as `ReleaseData` or `ProbeAckData` to the memory-facing C channel rather than CPU-facing D channel.

Because the writeback and request logic is responsible for C channel messages for their cache lines, the release handler is very simple. It only ever deals with `Release/ReleaseData` messages. It simply writes the data back to the SRAM if necessary and responds with `ReleaseAck`.

It is possible to have the request handler and the release handler handle multiple transactions in parallel to increase the throughput while waiting for responses. For example, a new `AcquireBlock` request (to an independent address) can be accepted while a previous one is blocked on a `ProbeAck` message. This parallelism is exposed as a “number of transaction trackers” parameter.

5.2.5 Testing

I tested Muntjac using the `riscv-tests` [101] testsuite provided by the RISC-V foundation. Dr Daniel Bates added the support as a GitHub action so each push can be tested automatically; he also added `riscv-dv` [45] for random test generation, to improve test coverage.

Dr Daniel Bates also implemented traffic generator, code coverage and testing support for TileLink IPs.

Apart from the regression and unit tests, I performed regular ad-hoc system tests, which involve tasks such as booting Linux (with Debian userspace) and running parallel benchmarks like PARSEC [17]. The FPU is tested with the `TestFloat` tool [61].

	Frequency	**Pipeline**		**Core**	
	(MHz)	LUTs	Registers	LUTs	Registers
FPU on	89	13663	4538	17420	6683
FPU off	97	6022	3098	9902	5266
FPU off, FP registers on	95	6111	3121	9924	5281

Table 5.3 Muntjac synthesis result for Xilinx Kintex 7 with Synopsys Synplify

I tested Muntjac on the Digilent Genesys 2 board (with Xilinx Kintex 7) and Nexys A7 board (with Xilinx Artix 7). The result for Xilinx Kintex 7 is shown in Table 5.3. Synopsys Synplify is used for its better ability to perform retiming compared to Vivado. With all cores including FPUs, the Genesys 2 board can comfortably fit a 4-core system and the Nexys board a 2-core system. Table 5.3 also shows that including the FP registers but not the FPU itself has a minimal impact on area, compared to the configuration where the floating point support is fully off. This configuration is a good option for running Linux distributions where code is compiled for RV64GC targets but floating point computations are not heavily used.

5.2.6 Performance Characteristics

Table 5.4 lists the approximate number of stall cycles an instruction will cause. All instruction and data caches accesses are assumed to hit. As described in Section 5.2.2, Muntjac backend has two execution stages and therefore can usually hide the latency of two cycles instructions. For example, memory loads or bitcasting a float to integer would create no stalls. It should be noted though these are still two cycle instructions, so if the succeeding instruction needs to use their output register, 1 cycle stall will be induced due to the read-after-write (RAW) data hazard. Only integer instructions from the base instruction set (RV64I) execute in a single-cycle.

For the Dhrystone and CoreMark benchmarks, I use the default configuration parameters. The L1 data and instruction cache are 16 KiB each, 4-way associative. The L1 data and instruction TLBs are 32 entry each and 4-way associative. The L2 cache is 64KiB/core in size and also 4-way associative. Both programs were compiled with GCC 9.2.0, with optimisations enabled and jump target alignment fine-tuned (GCC command line used

Instruction Type	Stall Cycles	Notes
Integer Arithmetic	0	No RAW data hazard
Load Store/Atomics	0 0/1	 1 cycle stall if succeeded immediately by a load or atomic
Integer Multiplication	3/10/16 (Slow Multiplier) 1/3/4 (Fast Multiplier)	Numbers in MULW/MUL/MULH order
Integer Division	33/65	33 for DIVW/REMU, 65 for DIV/REM
FP Arithmetic FP Division/Sqrt FP Bitcasts/Classification FP Comparison FP Misc	4 57 0 1 2	Same for FADD/FSUB/FMUL/FMA No separate treatment for single precision Includes sign manipulation, conversion
Jump/Branch (Predicted) Jump/Branch (Mispredicted)	0/1 3/4	+1 cycle stall if jump target is a misaligned 4-byte instruction.
CSR Access	2	Pipeline is flushed before access happens.
FENCE.I/SFENCE.VMA/ERET Traps Interrupts	7/8	Implemented as a pipeline flush followed by a redirection (similar to a mispredicted jump/branch).

Table 5.4 Number of stall-cycles for different instructions

were `-falign-functions=4 -falign-jumps=4 -falign-loops=4`). Muntjac achieves Dhrystone score of 2.17 DMIPS/MHz and CoreMark score of 3.01 CoreMark/MHz.

Memory Latency Testing

I performed memory access latency testing of Muntjac using Linus Torvalds' test-tlb tool [115]. The test is performed on a Genesys 2 board with 4 cores instantiated with default parameters, so has a total L2 of 256KiB. The DDR3 memory on the development board is used, with an approximate latency of 30 cycles to retrieve a cache line of data.

Although the testing environment is on an FPGA and not typical, I believe that the result is scalable and can be used to represent the performance of Muntjac if it were to be taped out and used in a real-word system. The most readily available DDR5 UDIMM modules available on market today [36] have transfer speed of 4800 MT/s (i.e. 2400 MHz clock) and CAS latency of 40 cycles, which corresponds to a first-word latency of 16.7ns and fourth-word latency (4 words $\times$ 2 channels would be a cache line) of 17.3ns. If we equate 17.3ns to 30 clock processor cycles (time taken to retrieve a cache line from the DRAM on the FPGA board), it would correspond to a not unreasonable core clock frequnecy of 1.7GHz.

Figure 5.16 shows the test results, with the memory access latency plotted against size of working set. The test was conducted with both sequential and random access patterns. It can be seen from the sequential plot that the latency for an L2 hit is approximately 20 cycles. The L2 miss latency is slightly more than 50 cycles (L2 hit penalty + memory access latency + some additional overhead). The latency of a randomly generated access pattern is,

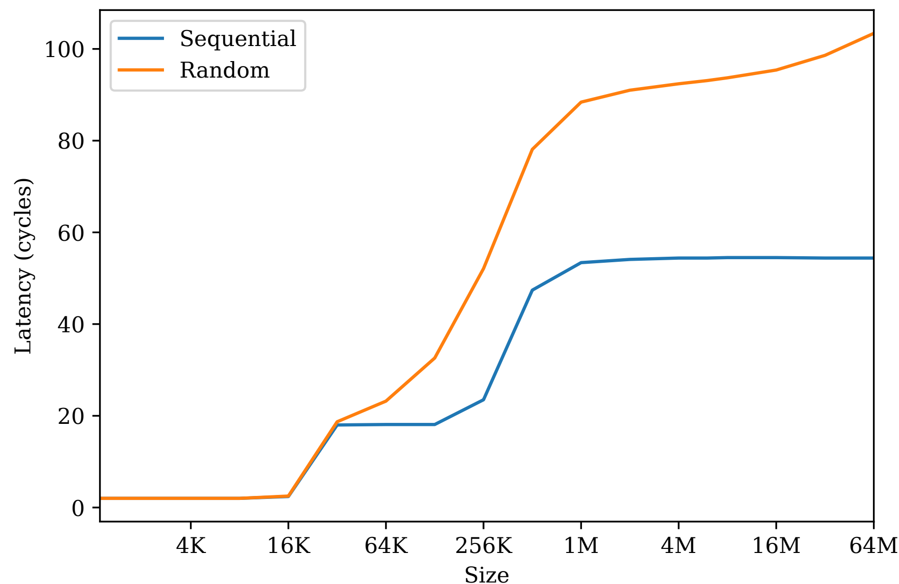


Figure 5.16 Memory access latency vs working set size

as expected, higher than that of sequential access due to additional time spent on page table walking upon TLB miss.

Another important property to understand is the core-to-core communication latency. This number has a deep impact on the performance of a wide range of synchronisation and inter-thread communication primitives. To measure this latency, I created a tiny benchmark tool [50] that creates two spin-based semaphores and let two threads up and down them alternatively, serialising the two cores. Care is taken to ensure that the two semaphores reside in separate cache lines to remove the effect of false sharing.

```
.L1:
    lw   t0, (a0)
    beqz t0, .L1
.L2:
    lr.w t0, (a0)
    beqz t0, .L1
    addi t0, t0, -1
    sc.w t0, t0, (a0)
    beqz t0, .L2
```

Listing 5.1 Assembly sequence for the `DOWN` operation of a semaphore that spins in the S state

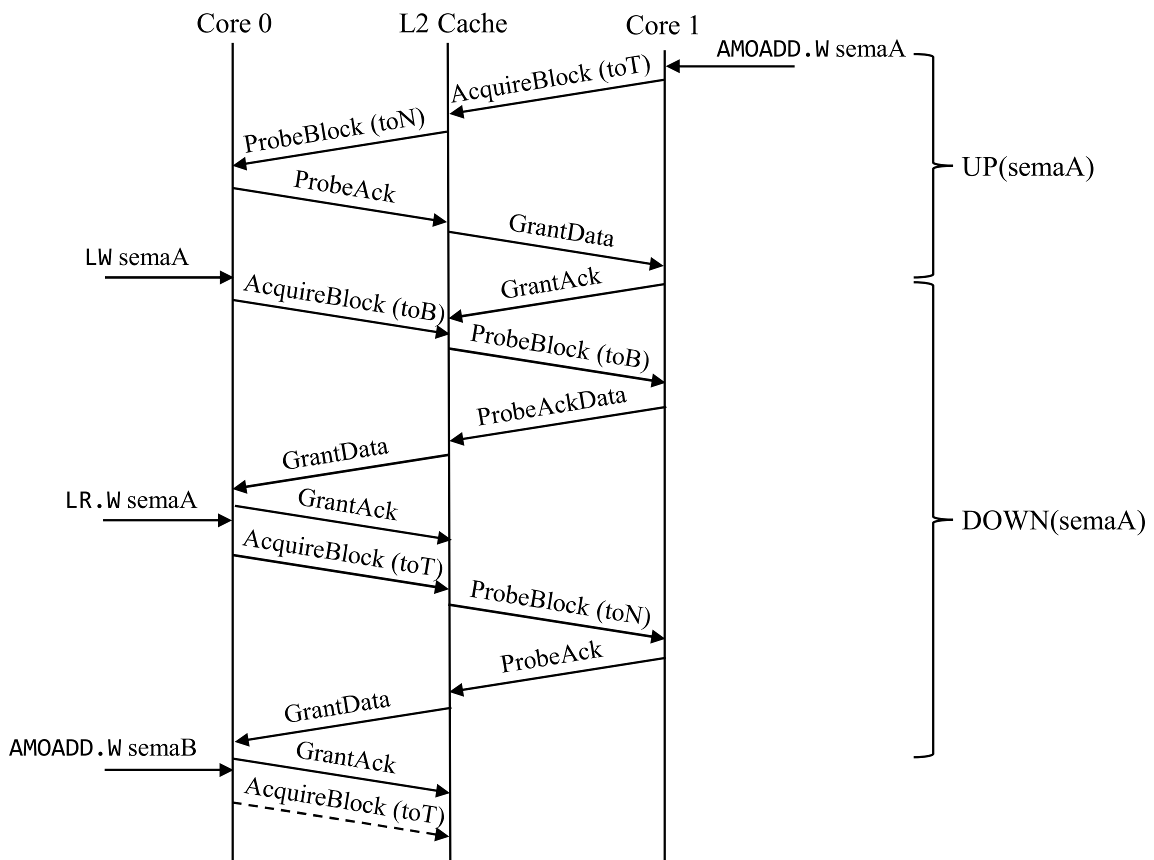


Figure 5.17 Bus transactions when a semaphore is DOWNed by core 0 and UPed by core 1, when spinning in performed in the S state

The latency between a core `UP` the semaphore and another core `DOWN` it is measured to be around 70 cycles (including overheads of control flow). An investigation reveals that these 70 cycles consist of 3 bus transactions that each compose of 4 serial bus messages (`GrantAck` is asynchronous), as shown in Figure 5.17. It can be seen that two bus transactions are needed to complete the `DOWN` operation. This corresponds to the two different type of load operation in the assembly sequence Listing 5.1. The spinning part of the semaphore is performed in the S state (corresponding `B` permission in TileLink) instead of the M state (`T` permission in TileLink). It is normally desirable to spin in the S state so that multiple cores can spin on the same semaphore without generating bus traffic.

```
1 .L1:
2     lr.w t0, (a0)
3     beqz t0, .L1
4     addi t0, t0, -1
5     sc.w t0, t0, (a0)
6     beqz t0, .L1
```

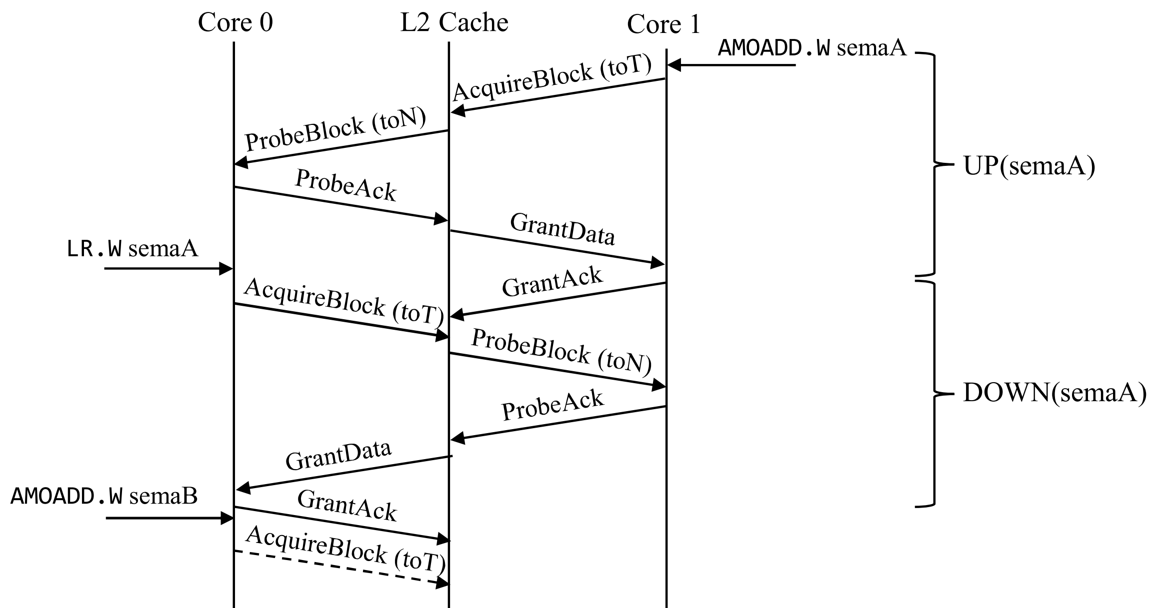


Figure 5.18 Bus transactions when a semaphore is `DOWN`ed by core 0 and `UP`ed by core 1, when spinning in performed in the M state

This however is not necessary when a semaphore is shared between only 2 cores; when a semaphore that spins on the M state is used (with assembly sequence Listing 5.2), only a

single bus transaction is needed for the LR/SC sequence, and the synchronisation can finish in two bus transactions, indicated in Figure 5.18. The latency of Muntjac with this type of semaphore is measured to be 44 cycles. For both spinning strategies, the time taken for each transaction is similar and estimated to be 20 cycles for all 4 serial messages. This is similar to the latency of an L2 hit indicated in Figure 5.16.

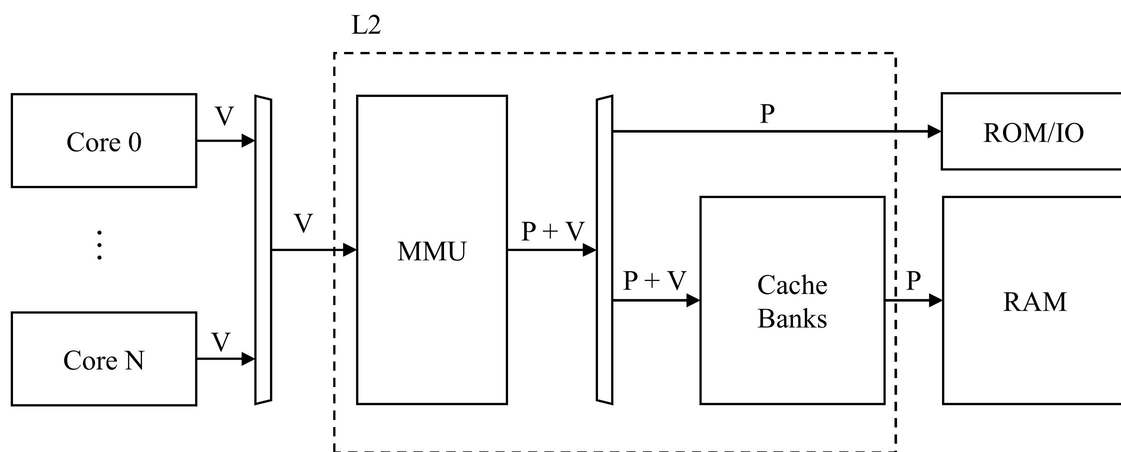


Figure 5.19 High-level component overview of the implementation of the virtual cache protocol

Figure 5.19 illustrates the major components involved in the implementation of the proposed virtual cache protocol. The L2 is decoupled into two parts, an MMU and one or more L2 cache banks. Each link in the diagram is marked with either `P`, `V` or `P+V`. `P` stands for the unmodified TileLink protocol which uses physical addresses. `V` stands for the proposed virtual cache coherence protocol. `P+V` is a special kind of link that only exists between the L2 MMU module and the L2 cache banks; it conveys both physical and virtual addresses, and can be easily converted into the vanilla TileLink protocol if the `B` channel is unused (the case for ROM and I/O accesses).

During the process of implementing the protocol in hardware, some issues that were masked by the higher-level simulation of the original protocol emerged. Fortunately none of them pose fundamental challenges; they have been addressed and the proposed protocol has been iteratively improved.

5.3.2 Dirty Cache Lines of Global Pages

In Section 4.6, I discussed that for a dirty cache line to be written back to the correct address, its physical address must be available to the L2. The problem described and addressed in that section is that after an ASID change (process switch), the root page table number of the inactive ASID (of the old process) is unavailable, making re-walking the page table to obtain the physical address impossible.

The first solution outlined in that section is to flush non-global entries from the L1 cache when the active ASID changes. Cache lines from global pages are excluded from the flush due to the assumption that global pages will have a consistent mapping across all ASIDs (hence the name global): because global pages will reside in all root page tables, we can use any ASID to initiate a page table walk, and the result will be identical. It turns out that this assumption is incorrect in practice.

In Linux, for each address space, the lower half of the address space are process-specific, user-space pages. The upper half are shared, global, kernel pages. There is a special process, the `init` process, the first user-space process loaded by the kernel. The Linux kernel uses the `init` process's page table as the "source of truth" for all kernel pages. When a new address space is created, the root PTEs of the kernel address range is copied from the `init` process.

Due to multi-leveled nature of page tables, the second-level page table of the global pages are shared, as shown in Figure 5.20. With this sharing, modification of non-root-level page tables can be done on the page table of the `init` process alone, and the change will automatically be reflected in other page tables. No special handling is necessary for this kind of modification – it behaves no differently from a page table modification of non-global, user-space pages.

The problem arises when a root PTE associated with the kernel address range is changed. This is a very rare circumstance – this happens only when the kernel requires additional virtual addresses to map kernel-space pages. For a 3-level page table like RISC-V's SV39, each root PTE corresponds to 1GiB of virtual address space. For Linux, it will only modify the mapping of the `init` process's root page table. It will leave the entry of the other root page tables invalid. When kernel-space code accesses relevant addresses in the context of a non-`init` process, a page fault will occur, and the page fault handler will only then lazily update the root PTE.

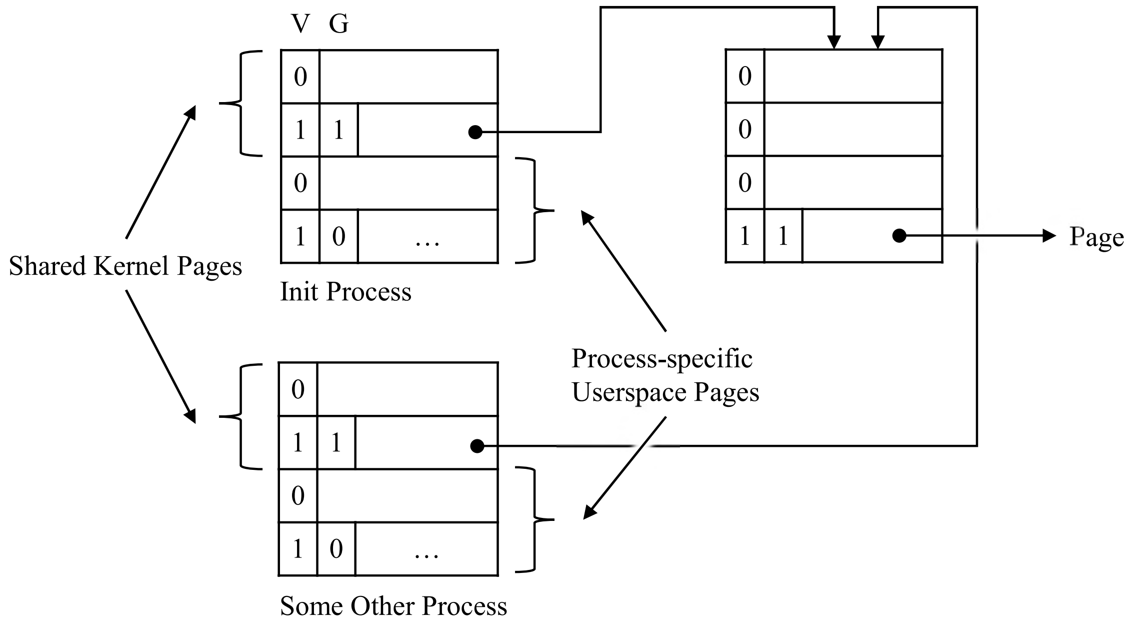


Figure 5.20 Representation of global pages in root page tables under normal circumstances in Linux

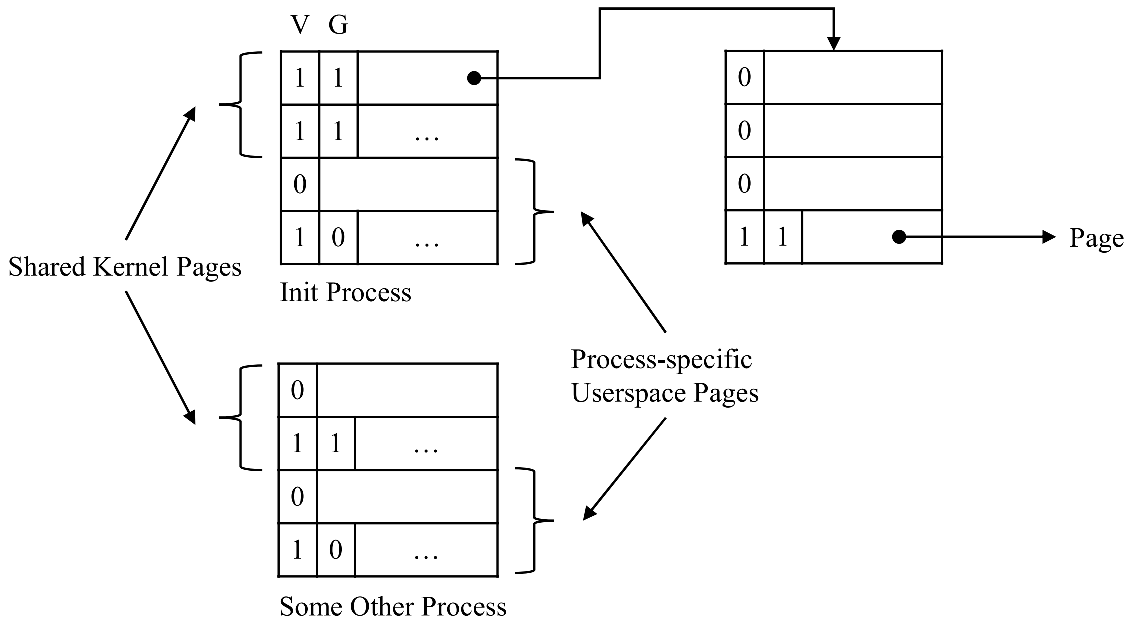


Figure 5.21 Representation of global pages after root PTE is changed in Linux

The lazy update approach means that there will be moments in the system where a global page is valid in one ASID and invalid in another, as shown in Figure 5.21. This is okay for physical caches, but in my proposed protocol, a dirty cache line of a global page can be retrieved when the address is valid in the active ASID, but be written back in another ASID, in which the address is not valid.

Luckily this problem can be easily solved at the OS level. I defined an atomic version counter for each process. The version counter for the `init` process will be incremented whenever kernel mapping at the root page table is changed. Whenever an address space switch happens, counter of the target process will be checked against the `init` process's counter; if the version is different, then relevant root PTEs of the target process will be copied from the `init` process.

One caveat is that the counter must be checked against the `init` process's version, not just the current process's. This is still related to the lazy behaviour of the kernel. When the kernel creates a mapping at the root page table, it is created in the `init` process's page table, and when it accesses it with the current address space, a page fault will happen, and the lazy code path will copy just one root PTE to mitigate the fault. Since the "effective" version of a root page table can be newer than the stored version, the check before switching must use the latest version, which is the `init` process's counter.

It should be noted that the global page issue described in this section still exists even if the second solution to the dirty cache line writeback problem were to be adopted (the extra level of indirection option as described in Section 4.6). It could however be more easily dealt with by assigning the `init` process a special ASID, and then let the memory management hardware use the root page table associated with that ASID for performing page table walks dirty cache lines of global pages were to be written back, instead of using active ASIDs.

5.3.3 Exclusive Cache Lines and Page Table Entry Changes

For a cache line with physical address p, $r(p)$ is an overestimate, so if $v \in r(p)$, it is possible that $f(v) = p' \neq p$. If at this stage the L2 cache tries to invalidate p from an L1 cache, and the L1 holds a dirty cache line corresponding to address v, then it is possible that the dirty data is written to p rather than p' – this both causes the data in p to change unexpectedly and dirty data to p' be lost.

Given that our overestimation of r is usually represented in a wildcard state, which only happens when a synonym is involved, all related cache lines will be read-only in this scenario, so there will be no dirty data to writeback, so the aforementioned problem is not an issue. However, during the implementation and validation process it was discovered that

the proposed virtual cache protocol does still suffer from this problem without presence of synonyms, when changes to PTEs are involved.

One example scenario where this overestimation property can lead to problems is the following: if there is a page table change so the address translation function f changes to f' where $f(v) = p$ and $f'(v) = p'$. If before this page table change, this cache line is in exclusive state in an L1 and not dirty, then we will record $r(p) = v$. When the PTE is changed, as Section 4.5 discussed the L1 will have to be flushed to remove v from it. Because v's cache line is not dirty, this cache line does not have to written back and can be unilaterally flushed, so the L2 can still consider $r(p) = v$ and the previous L1 as the owner of this cache line.

After the page table change, if an L1 accesses this cache line again, the L2 will record $r(p') = v$. In the current state, if the cache line p is accessed again, because it still considers the previous L1 as the owner, it will send an invalidation with address v. If writeback triggered by this invalidation message is considered as a reply to the invalidation of p, the dirty data will be written to p instead of p'. This both causes the data in p to change unexpectedly and dirty data to p' be lost.

To address this problem, I modified the way `ProbeAckData` messages are handled. Instead of just forwarding it to the handler that issues the `Probe` message, the message goes through the normal address translation routine, i.e. TLB and PTW, to find the target address to writeback. If the physical address found is different from the address expected by `Probe`, it is broken up into a `ReleaseData` message followed a `ProbeAck` message. It should be noted that it is legal to break all `ProbeAckData` into `ReleaseData` + `ProbeAck`, but I choose to only do this when addresses are different to reduce the associated performance penalty.

5.3.4 Dirty Cache Lines and Page Table Walking

One other problem noticed with the originally proposed protocol is a deadlock scenario as depicted in Figure 5.22. If a TLB access triggered by a `ReleaseData` message misses, a page table walk will happen; the PTW will issue `Get` messages to the L2 to read PTEs needed. If a cache line containing the required PTE happens to be owned by an L1, then deadlock can arise: the L2 generates a `Probe` message, but the L1 is waiting for a `ReleaseAck`, so it cannot reply to this message (otherwise it will need to support arbitrary level of nested replies, which is not practical in hardware). Similarly, a `ProbeAckData` message may also trigger a recursive `ProbeBlock` as it also requires a proper address translation (as described in the section above), and therefore can risk deadlocks.

The reason for this deadlock is obvious: in TileLink it is required that a message causes a TileLink agent to create a new message, the new message must be in a channel with the same or higher priority – in this deadlock scenario, `Get` is in the lowest-priority channel A, while

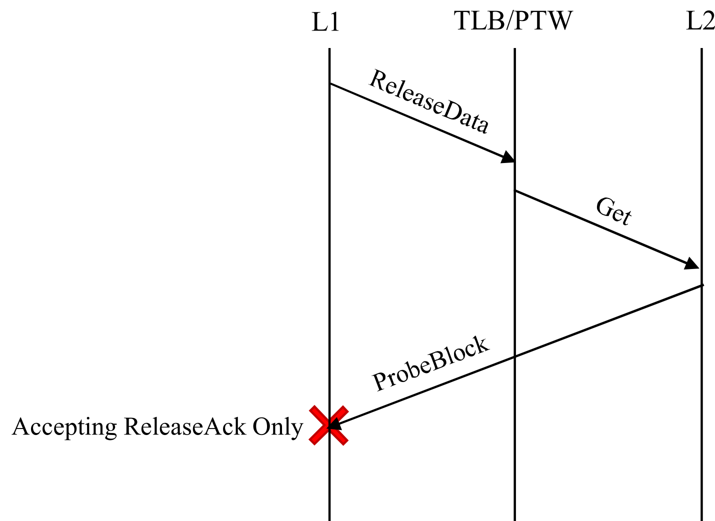


Figure 5.22 Protocol transactions when a page table walk happens for a writeback message

ReleaseData is in the channel C; the inversion of priority violates the assumption used in TileLink to establish deadlock-freedom.

I addressed this issue by introducing a new type of message GetNonblocking. This message is largely identical to the Get message, except that it is issued on channel C instead of A. Moreover, when the L1 receives this type of message, it will simply serve the copy it has (or in case it has none, forward the message to the next-level cache in an uncached manner, without causing eviction and refills). The data served this way could be stale, but it is okay because the data corresponds to a dirty page – in Section 4.5 we have already established that the OS requires to flush corresponding dirty cache lines before a mapping can be changed, so the PTE definitely has not been unmapped or remapped between the acquisition of the dirty cache line by the L1 and the writeback. Page attributes, like accessed or dirty bits, may have been changed in the interim, but we can still get the correct physical address for writeback in this case.

It should be noted that for a AcquireBlock message, if the TLB misses, the triggered message should still be Get so it can obtain the up-to-date PTE – mapping a new page, unlike unmapping or remapping, does not require a cache flush or a TLB flush.

5.3.5 Dirty Cache Lines and the Accessed Bit

RISC-V's page table attribute contains an A accessed bit and a D dirty bit. There are two schemes permitted to manage these bits:

1. A page-fault exception is raised when the A bit is clear, or when the D bit is clear and the access is a write.

2. Hardware atomically sets the `A` bit if the page is accessed, and sets the `D` bit if it is written to.

The baseline Muntjac implementation chooses option 1 and leaves the OS to manage these bits. It simply processes the `A` bit together with the `V` valid bit, and the `D` bit together with the `W` writable bit. My virtual cache implementation takes the same approach.

This creates a problem because Linux sometimes clears the `A` without flushing the cache first (because the page is neither unmapped nor remapped). In a very rare circumstance, this `A` bit clearance can happen to a page where a cache line within it is dirty in an L1 but the translation of this page has been evicted from the TLB. When the dirty cache line is written back, the MMU unit will consider the page invalid and enter an error condition.

The solution to this caveat is simple: when the MMU unit is dealing with a writeback, and the `A` bit is clear but `V` bit is set, it will still consider the page to be valid and perform the writeback. However, for such pages, the result of page table walking will not be inserted back into the TLB to prevent the TLB entry from being used by `AcquireBlock` or other requests.

5.3.6 Signal Descriptions

The proposed protocol does not add additional channels to the TileLink protocol. The same five channels are used, where *A*, *C*, *E* flow from host to device, and *B*, *D* flow from device to host. All flow control and channel prioritisation rules are respected, so the deadlock freedom guarantee provided by the TileLink protocol can be preserved and inherited. Signals within each channel are tweaked to suit the demand of the protocol.

Name	P	V	P+V	Description
opcode	✓	✓	✓	Indicates the message type.
param	✓	✓	✓	Parameter specific to a given message type.
size	✓	✓	✓	$\log_2$(size in bytes).
source	✓	✓	✓	Source identifier of the requester.
address	✓		✓	Target physical address of this message.
mask	✓	✓	✓	Byte mask for data.
data	✓	✓	✓	Data to transfer.
vaddr		✓	✓	Target virtual address of this message.
asid		✓	✓	ASID associated with the virtual address.
ptn		✓		Root page table number associated with the ASID.
attr			✓	Page attributes associated with the virtual address.

Table 5.5 Signal lists of the `A` channel

Name	P	V	P+V	Description
opcode	✓	✓	✓	Indicates the message type.
param	✓	✓	✓	Parameter specific to a given message type.
size	✓	✓	✓	$\log_2$(size in bytes).
source	✓	✓	✓	Source identifier of the receiver.
sink	✓	✓	✓	Sink identifier of the sender.
denied	✓	✓	✓	Indicate if the request results in error.
data	✓	✓	✓	Data to transfer.
attr		✓	✓	Page attributes associated with the request.

Table 5.6 Signal lists of the `D` channel

Name	P	V	P+V	Description
sink	✓	✓	✓	Sink identifier of the receiver.

Table 5.7 Signal lists of the `E` channel

Take an `AcquireBlock` message as an example, this message is carried on `A` channel, and its response `GrantData` is carried on `D` channel. Table 5.5 shows that for the proposed virtual protocol, the virtual address is carried in channel A instead of the physical address; it also carries the necessary information to perform address translation. The MMU unit in the L2 performs the translation, discards the root page table number which is no longer necessary, and carries the physical address and page attributes down to the L2 cache bank. If the L2 cache bank accepts this request, it sends a `GrantData` message on the `D` channel. `D` channel in the proposed protocol is not very different from the original TileLink protocol; as shown in Table 5.6, it is only extended with an attribute signal, so the L1 cache can enforce necessary privilege checks. `D` channel is untouched by the MMU unit and is forwarded directly back to L1 caches. `GrantAck` message on `E` channel is same as TileLink as shown in Table 5.7. Uncached access transactions (`Get`/`AccessAckData` and `PutFullData`/`PutPartialData`/`AccessAck`) are similar.

TileLink voluntary cache line writeback is done by `ReleaseData` and `ReleaseAck` messages, on `C` and `D` channels respectively. The `C` channel in the proposed virtual cache coherence protocol is modified in a similar way to `A` channel, as shown in Table 5.8, where MMU unit translates the address from virtual to physical. Page attributes are not relevant for the writeback, so they do not exist in `C` channel. The page attribute signal in channel `C` used by `GrantData` messages is unused in `ReleaseAck` messages.

Invalidation is more complex due to the existence of batch invalidations and the issue mentioned in Section 5.3.3. Apart from virtual address and ASID tagging that appears in

Name	P	V	P+V	Description
opcode	✓	✓	✓	Indicates the message type.
param	✓	✓	✓	Parameter specific to a given message type.
size	✓	✓	✓	$\log_2$(size in bytes).
source	✓	✓	✓	Source identifier of the requester.
address	✓		✓	Target physical address of this message.
data	✓	✓	✓	Data to transfer.
vaddr		✓	✓	Target virtual address of this message.
asid		✓	✓	ASID associated with the virtual address.
ptn		✓		Root page table number associated with the ASID.

Table 5.8 Signal lists of the `C` channel

Name	P	V	P+V	Description
opcode	✓	✓	✓	Indicates the message type.
param	✓	✓	✓	Parameter specific to a given message type.
size	✓	✓	✓	$\log_2$(size in bytes).
source	✓	✓	✓	Source identifier of the receiver.
address	✓		✓	Target physical address to invalidate.
vaddr		✓	✓	Target virtual address to invalidate.
asid		✓	✓	ASID associated with the virtual address.
mask		✓	✓	Indicate the type of batch invalidation. Encoded together with `asid` and `vaddr` as in Figure 4.11.

Table 5.9 Signal lists of the `B` channel

both `A` and `C` channels, a `mask` field also exists in `B` channel, as shown in Table 5.9. The `mask` field is used by `ProbeBlock` message to indicate the type of batch invalidation. It is encoded together with the `asid` and `vaddr` field, using the scheme described in Figure 4.11. As a recap, if it is set to 1, it means that both `asid` and `vaddr` are valid and this is a normal (non-batch) invalidation. If `mask` is 0 and `asid` is 1, `vaddr` is valid and this is an ASID-wide invalidation. If `mask` is 0 and `asid` is 0, then this is a full batch invalidation or a partial batch invalidation depending on the value of `vaddr`.

For responses to normal invalidation, `ProbeAck` or `ProbeAckData` messages are sent on `C` channel and is processed similar to `Release`/`ReleaseData` messages. For responses to batch invalidation, `ProbeAckData` is disallowed – since under the proposed protocol synonym is only allowed to occur in read-only mode (where L1 caches only have `B` permission, i.e. in S state), hosts are not allowed to perform writeback in response to batch invalidation. Instead of sending multiple messages per cache line invalidated, the host sends a special type of `ProbeAck` message on the `C` channel. MMU does not perform address translation for this kind of `ProbeAck` message.

It should be noted that the physical address field exists in the `P+V` variant but not in the `V` variant. This does not indicate that the MMU unit simply drops it. Instead, both physical and virtual addresses of pending invalidations are stashed by the MMU unit when it sees it. For `ProbeAckData` messages, if the physical address after address translation differs from the stashed physical address, the message is broken up into `ReleaseData` request with actual physical address and `ProbeAck` message with the stashed physical address, as described in Section 5.3.3.

The proposed protocol also adds a few messages. `GetNonblocking` on `C` channel is added as per Section 5.3.4. This message is allocated with a new opcode. Given that the only issuer of this message is the page table walker inside the MMU unit itself, and it will not be issued by the L1 caches, it does not require processing by the MMU unit. The response to `GetNonblocking`, `AccessAckDataNonblocking` is sent on channel `D`. Given that opcode is 3-bit and there are already 8 types of `D` channel messages being used, instead of allocating a new opcode for this type of message, the same opcode as `AccessAckData` is used. To disambiguate the two types of messages, I choose to represent the nonblocking variant using non-writable, non-readable and non-executable page attribute (an `Get` message that results in such attributes will be denied directly by the MMU unit and will not continue to the L2 cache banks).

Another message being added is `TlbFlush`. I choose to encode this message as a subtype of TileLink's `Intent` message (two subtypes defined in the original TileLink specification are for read/write prefetches). The response to this message is `HintAck`, same as replies

to other Intents. A TLB flush in the proposed virtual cache protocol requires a special

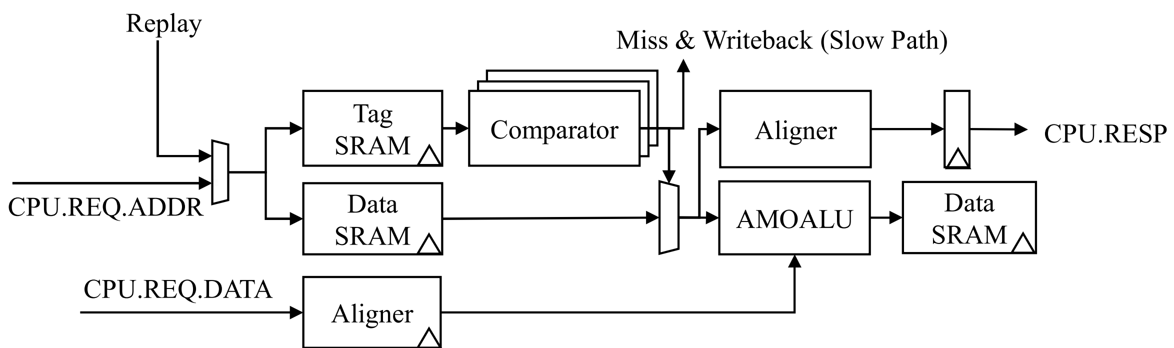
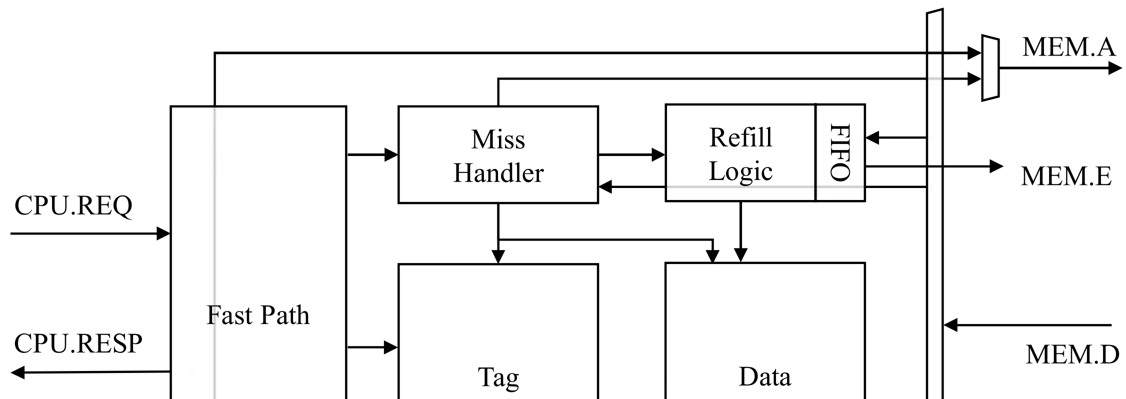


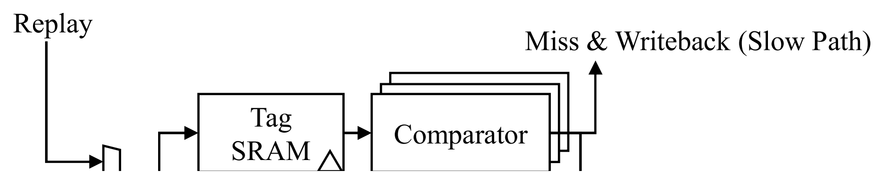
Figure 5.23 Fast path of the virtual L1 data cache design

The data cache is derived from Muntjac's data cache. The TLB is removed from the fast path, as shown in Figure 5.23. The comparator within the fast path is modified to check virtual addresses with ASID tags instead of physical addresses. The comparator additionally needs to verify the user/supervisor mode bit of the page attribute copy in the metadata matches the running mode.

The miss handling logic, refill logic and writeback logic as shown in Figure 5.24 are minimally modified. The probe logic is modified to support batch invalidations as mandated by the protocol. When a batch invalidation is received, the probe logic will calculate the step size needed when scanning through the cache. The step size will be small for a full batch invalidation, and will be larger if more bits are specified in a partial batch invalidation. If the OS implements the co-alignment assistance as described in Section 4.7, then the scan step size will exceed the number of sets in the cache, therefore only require scanning one single associative set. Batch invalidation and normal invalidation do not share the same pathway even if only one set needs to be scanned, as batch invalidation will not invalidate exclusive or dirty cache lines.

The flush logic is modified to send the `TlbFlush` intent and wait for the `HintAck` response when the pipeline issues a `SFENCE.VMA` instruction, in addition to performing cache flush. The PTW component is removed with the TLB as it is no longer needed in the L1.





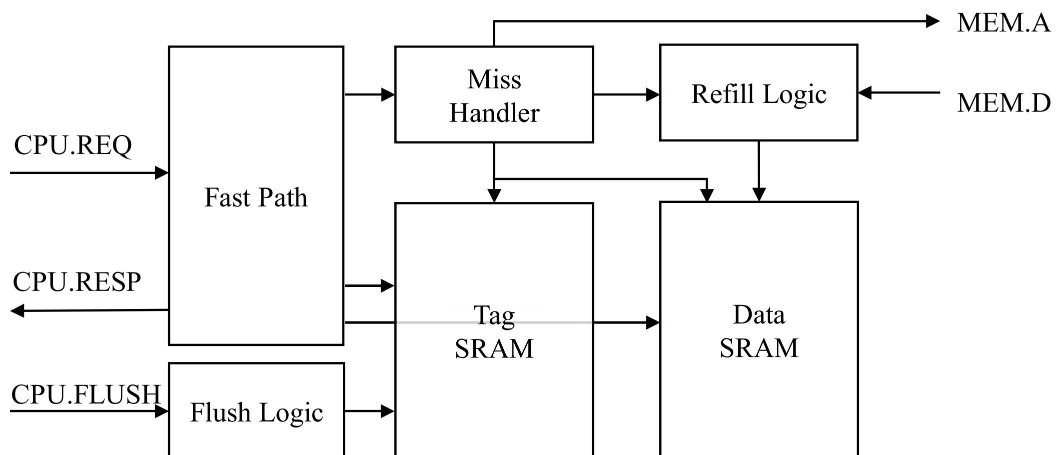


Figure 5.26 Components of the virtual L1 instruction cache design

Similarly, the instruction cache is derived from Muntjac's instruction cache. As shown in Figure 5.25 and Figure 5.26, there are less changes required compared to the data cache; no major changes are made apart from removing the TLB and PTW and modifying the comparator. Given that the instruction cache is non-coherent and communicates in TL-UH protocol instead, probe logic is non-existent in the instruction cache and it does not need to respond to invalidations. The flush logic also does not require making `TlbFlush` requests; simply flushing all contents from the cache is sufficient.

5.3.8 L2 Cache Design

The L2 cache is divided into two parts, an MMU unit and cache banks. An early implementation does not have this decoupling; during implementation it became apparent that this separation is necessary as different regions of physical addresses require different handling. In Figure 5.19 it can be seen that there is a demultiplexer at the device-facing side of the MMU, passing memory-like addresses down to cache banks and filtering out non-cacheable I/O accesses and read-only ROM accesses for separate handling. This demultiplexation needs to be performed on physical address and cannot be done with just virtual addresses, so it must happen after the MMU unit. An additional advantage of this decoupling is that multiple cache banks can be supported, multiplexed by physical addresses.

The MMU unit is depicted in Figure 5.27. The request handling logic and release handling logic operate independently from each other, with access to the TLB arbitrated. PTW needs to be invoked when a TLB access misses; I choose to have two separate PTWs, one for the request logic and one for the release logic. Using a single PTW is feasible; however as discussed in Section 5.3.5, page table walking during request and release are slightly different, and as discussed in Section 5.3.4, different messages are to be used when reading the page table (`Get` vs `GetNonblocking`), so separate PTWs can make implementation simpler. Both the request handler and the release handler are designed for high-throughput operation; they both operate at wire speed if the TLB access hits.

The request handler is also responsible for permission checking – if an `AcquireBlock` with a `NtoT` parameter (requesting transition into M/E state) and the requested cache line lies in a non-writable page, then the request handler will hand the message over to the exception handler, which will deny the request. Similarly, messages towards an invalid virtual address are denied within the MMU unit.

The release handler does not check for permissions, as its sole task is to translate the virtual address to the correct physical address for writeback. The accessed bit is also ignored as discussed in Section 5.3.5. For each `ReleaseData` message, the address is translated and the message is passed to the L2 cache banks. For `ProbeAckData` message, as described in

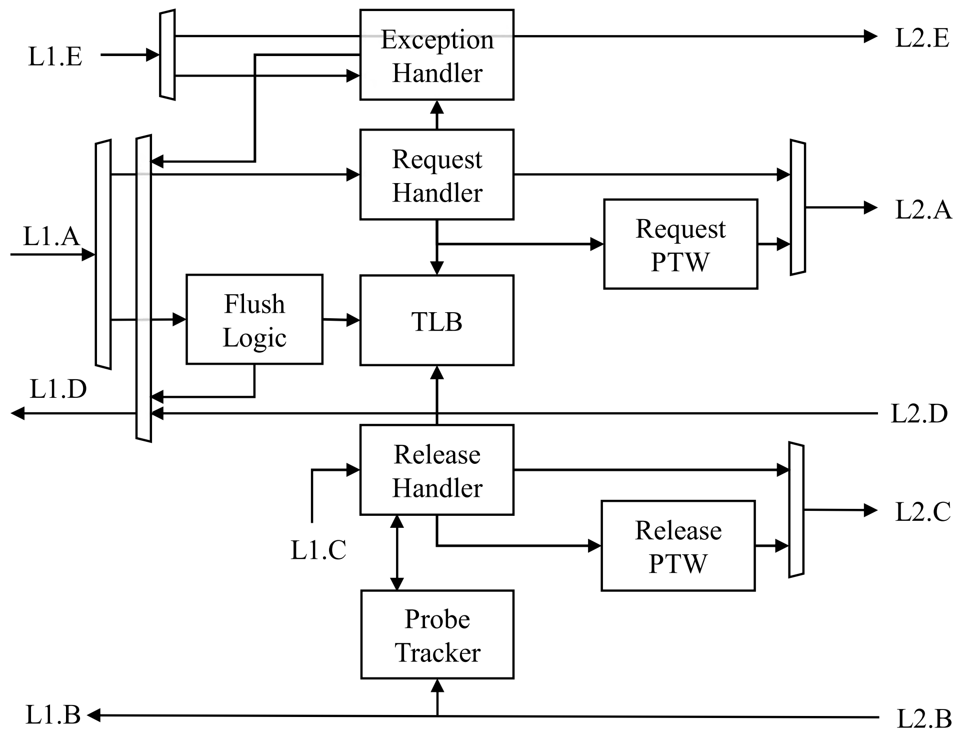


Figure 5.27 Memory management unit of the virtual L2 cache, responsible for virtual address translation and protection

Section 5.3.3, care must be taken when the physical address after translation differs from the physical address that triggers the invalidation. The dirty data must be written to the translated physical address but the physical address being invalidated needs to be notified to complete the invalidation. The MMU unit utilises a probe tracker to bookkeep the physical address that triggers an invalidation, because as Section 5.3.6 described, the L1 cache operates using virtual addresses only. When a `ProbeAckData` is received, two physical addresses are compared, and it is broken into `ReleaseData` followed by `ProbeAck` if the address mismatches, as described in Section 5.3.3.

`TlbFlush` messages are filtered out from the `A` channel and feed into the flush logic, which flushes the TLB and responds to the L1 with an `IntentAck`. Flushing takes priority to other TLB accesses. This type of message is handled entirely within the MMU unit and is opaque to the L2 cache banks.

Figure 5.28 shows the L2 cache bank design under the proposed virtual cache coherence protocol. At a component level it is similar to Muntjac's L2 cache design. The notable exception is the addition of nonblocking access logic, required by Section 5.3.4. Unlike the request handler, the data SRAM is accessed directly regardless of owners of the cache line. If the cache line does not exist in the L2 cache, then the data is read from main memory directly

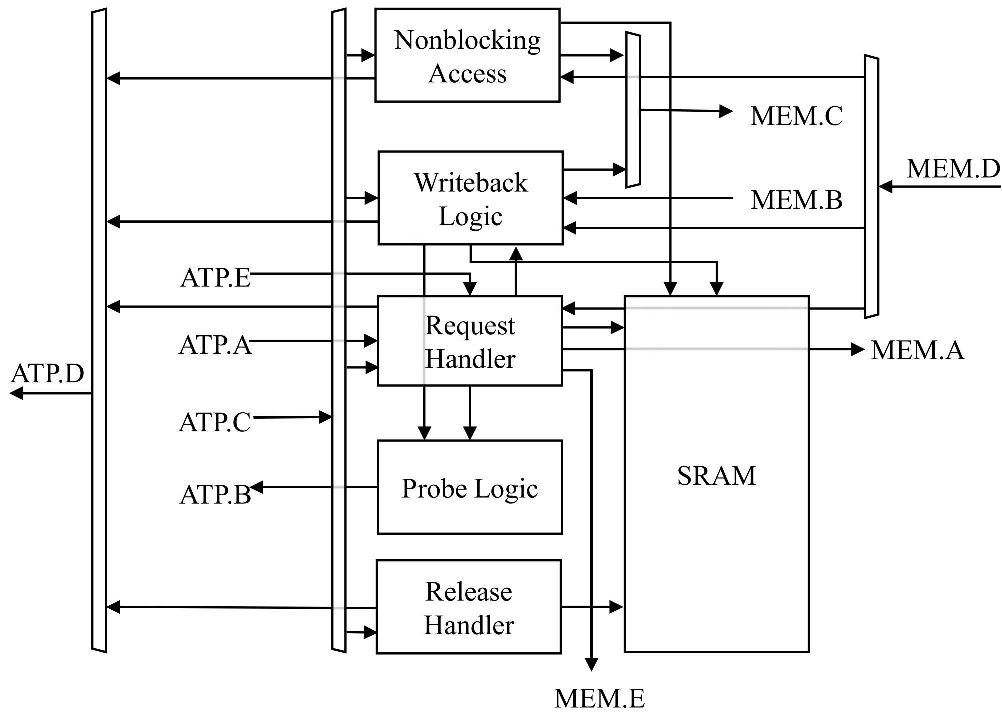


Figure 5.28 L2 cache bank design under the proposed virtual protocol

and doing so does not refill the cache line in the L2. Metadata fields are expanded and cache line state transition logic are modified in accordance with Section 4.8.

5.4 Evaluation

For evaluation, the baseline is a SoC with 4 Muntjac cores with the default cache configuration. The default cache configuration of Muntjac uses 16KiB, 4-way associative cache for both instruction cache and data cache, and 32-entry 4-way associative for both the instruction TLB and the data TLB. I configured the L2 cache to be 256KiB and have 4 trackers (i.e. able to handle 4 transactions in parallel). This matches both the configuration I used in the simulation in Section 4.4 and the configuration during Muntjac testing in Section 5.2.5. FPUs are instantiated for all cores as they are necessary for PARSEC benchmark.

For this section, all benchmarks are run on the FPGA. A RISC-V CLINT and a PLIC are provided. The SoC also have some I/O devices, including a ROM controller hosting firmware that handles the booting of the SoC and provides SBI, an UART 16550 for serial port, a SD card controller for accessing the root file system, and a network controller. The network controller is used for communicating with the host computer; for benchmarks the host computer sends the command to be run via the network and received the result back through

network. All results quoted in this section are collected when running Linux 5.15, with necessary patches applied to support the SoC and the evaluated cache coherence protocol, and an unmodified Debian userspace. I fixed the frequency for all configurations to 80 MHz, a frequency that all configurations can achieve with all the I/O devices, interconnects and FPUs enabled.

The virtual cache coherent configuration follows Section 4.4, and uses a 256-entry, 4-way associative unified TLB.

5.4.1 Utilisation

	Frequency	**Pipeline**		**Core**	
	(MHz)	LUTs	Registers	LUTs	Registers
FPU on	91	13833	4696	17055	6184
FPU off	107	5951	3113	9307	4679
FPU off, FP registers on	102	6130	3136	9430	4657

Table 5.10 Muntjac synthesis result for Xilinx Kintex 7 with Synopsys Synplify, with virtual cache

The utilisation of a Muntjac core with virtual cache is shown in Table 5.10. Compared to the physical baseline in Table 5.3, the virtual cache results in lower utilisation as the TLB is removed from the L1. The utilisation reduction is not significant though because the L2 cache requires an additional MMU unit and slightly more complex cache line state transition logic, which offsets the utilisation reduction associated with each core.

As a TLB access is required for each memory access, and it is accessed in parallel to cache line metadata fetch to reduce memory load latency, it is part of the fast path. Removing it from the critical path helps with timing, consistent with frequency increase observed in Table 5.10.

5.4.2 Memory Latency Testing

The same memory access latency tests used for Muntjac are applied on the virtually coherent version. The memory latency curve is shown in Figure 5.29. It can be seen that, for sequential accesses, the physical and virtual cache version behaves almost identically before the L2 cache is saturated and the requests need to be served from the main memory. The virtual cache version is 3 cycles slower than the physical baseline due to additional cycles required for address translation (address translation is parallel to the L1 cache lookup in the physical

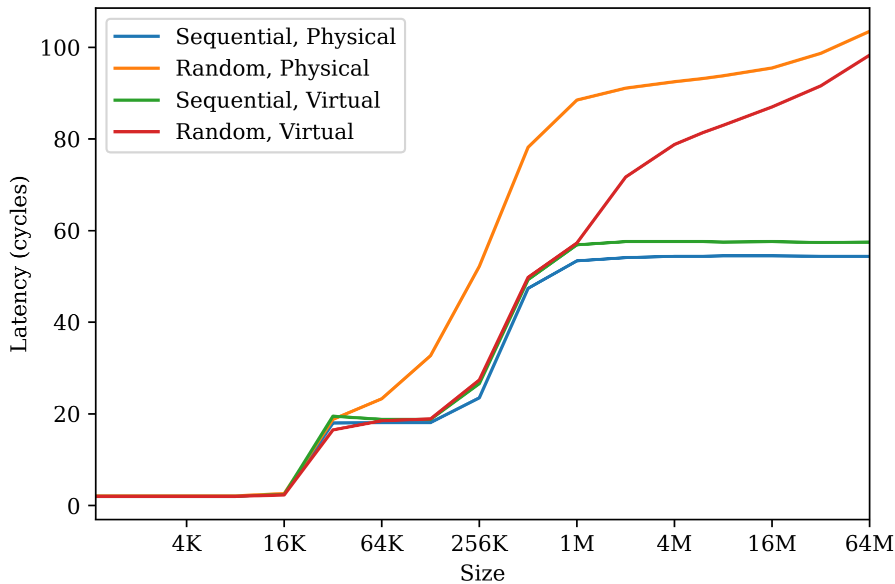


Figure 5.29 Memory access latency vs working set size

baseline). Due to implementation reasons the memory-facing link of the L2 cache in the virtual setup also have 1 cycle penalty compared to the physical cache version. Interestingly, for random accesses it is the virtual protocol that performs better. This shows one of the advantages of the proposed protocol – the TLB in the baseline is thrashed when working set size is increased; the virtually coherent version, due to a larger shared TLB, is able to handle a larger working set size before performance degradation.

The inter-core communication latency is also affected by the additional 1-cycle latency introduced by the L2 MMU unit. The latency for each semaphore synchronisation is measured to be 75 cycles compared to 70 cycles for the baseline when the semaphore spins in M state; the difference is 47 to 44 when the semaphore spins in the S state. The 1 cycle latency perfectly explains the cycle count difference measured (in Figure 5.17, each `AcquireBlock` and `ProbeAck/ProbeAckData` costs 1 extra cycles, except for the `AcquireBlock` parallel with `GrantAck`).

5.4.3 PARSEC Benchmark

The PARSEC benchmark, which is used to evaluate the performance of virtual cache in simulation, is also used to test the performance of the hardware implementation. Figure 5.30 shows the PARSEC performance of the hardware implementations. The Muntjac SoC with physical caches is selected as the baseline for numerical comparison. Apart from the virtual cache implementation, I included the two variants described in Section 4.8, the physical

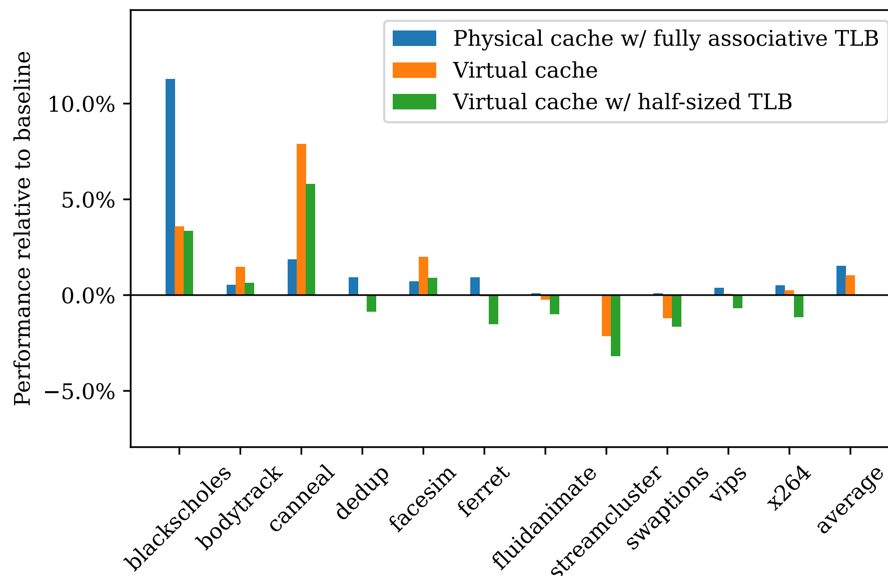


Figure 5.30 PARSEC benchmark performance comparison between physical, virtual cache implementations and their variants

version with a fully associative TLB and the virtual variant with a half-sized TLB. The overall performance metrics match those obtained through simulation in Section 4.4 (see Figure 4.19), but there are some significant mismatches for particular workloads, e.g. `canneal`'s performance is significantly better when evaluated using the hardware implementation vs the simulated protocol. I have not established the concrete reason for the difference, but one major difference between the simulation setup and the hardware setup is that L2 hardware supports 4 transactions in parallel while this is not modelled in the simulation (note that the 1-cycle delay described in the previous section is considered and accounted in the simulation).

On average, the virtual cache implementation performs 1.0% better than the baseline. This is less performant than the fully associative variant, which is 1.5% better than the baseline, unlike in the simulation where their performance are almost equivalent. Both setups show less improvement over the baseline compared to the result of simulation. The virtual half-sized TLB variant has equivalent overall performance (less than 0.1%) to the baseline, consistent with the simulation outcome.

The 1-cycle delay introduced by the MMU unit has a quite significant impact on the performance. If I introduce an artificial 1-cycle delay into the path between the L1 and the L2 caches in physical setups, then the virtual cache will have 2.7% improvement over the delayed baseline; this will be faster than the fully associative physical variant (also delayed), which is just 1.6% faster. Virtual cache, even with smaller overall TLB sizes, is 1.7% faster.

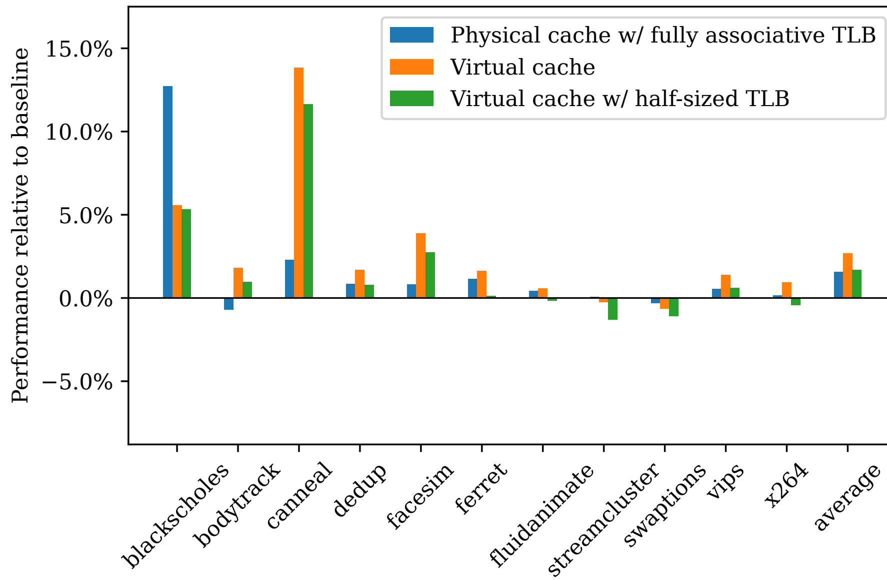


Figure 5.31 PARSEC benchmark performance comparison between physical, virtual cache implementations and their variants, with 1-cycle delay added to the physical caches

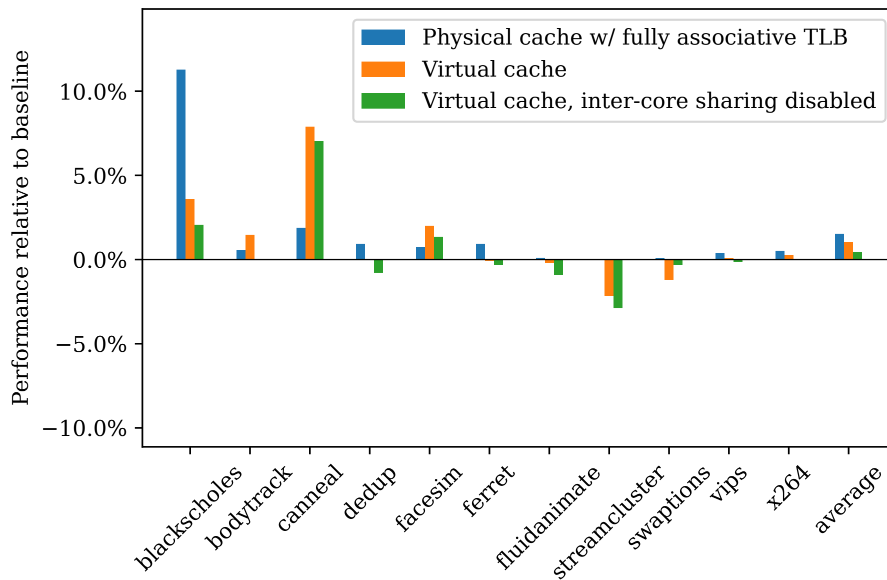


Figure 5.32 PARSEC benchmark performance comparison of virtual cache implementations, with inter-core TLB entry sharing disabled

The improved performance can be mostly attributed to consolidation of TLB in the proposed protocol, which facilitates sharing, as described in Section 3.3.1. Out of the sharing enabled, the most significant contributor is the possibility for instruction cache and data cache to share the same TLB, given that many workloads, including benchmarks in PARSEC, have a much larger working set size then size of hot code. Figure 5.32 shows that even with inter-core TLB entry sharing disabled (implemented as having 4 MMU units as described in Section 5.3.8, one for each core, with TLB size inside the MMU reduced to $\frac{1}{4}$ of the total), the virtual cache implementation can still have 0.4% improvement over the baseline (or 1.7% over the delayed baseline).

5.4.4 Minibench

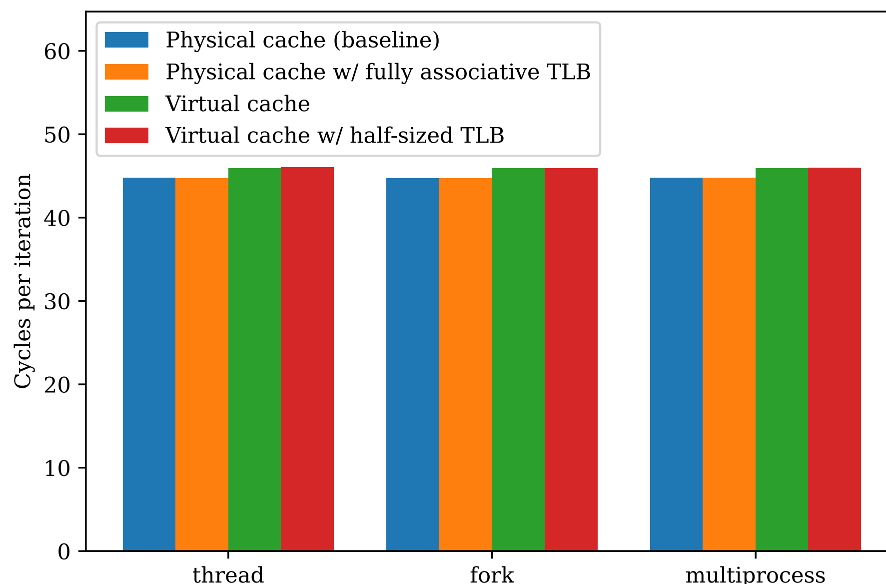


Figure 5.33 Summary of minibench evaluation on hardware implementation, with a `scan` run, size of 32768 bytes, stride of 64 bytes

Figure 5.33 and Figure 5.34 shows the result of minibench runs on the FPGA. The results closely match the simulation result presented in Section 4.8.

5.4.5 Conclusion

Overall, I believe the implementation of the proposed protocol is successful and meets the expectation and the initial design criteria:

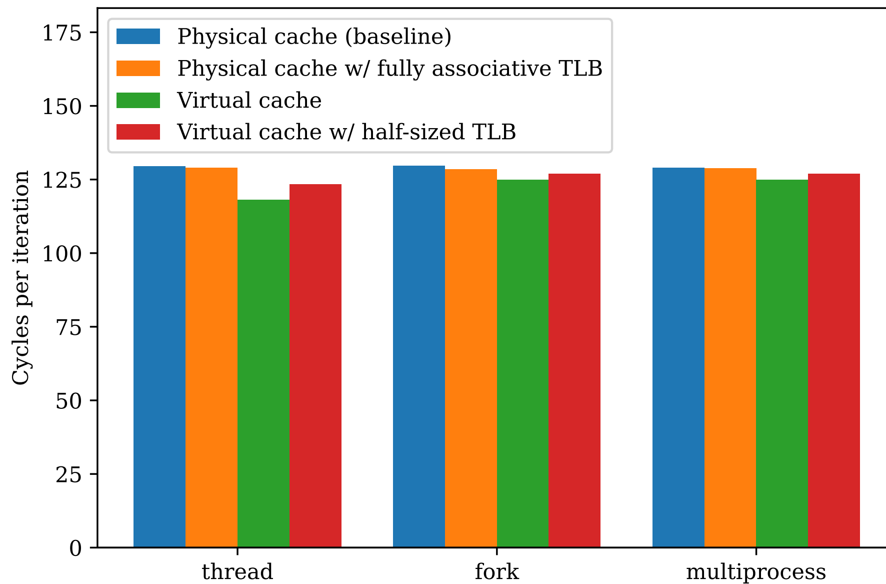


Figure 5.34 Summary of minibench evaluation on hardware implementation, with a `scan` run, size of 2097152 bytes, stride of 4096 bytes

- The utilisation is lower than the baseline for components that are closer to the pipeline. The critical path of the virtual cache is shorter and therefore provides a slight frequency advantage, while no performance loss and a small gain is measured compared to the baseline when running at the same frequency.
- The implementation is capable of running unmodified Debian userspace therefore has high compatibility with existing programming models.
- There are a few patches required to get Linux kernel working properly under the proposed protocol, but the porting process is relatively simple; the modification requires less than 1000 lines of code.

Chapter 6

Virtual Cache Coherency with Accelerators

Reflecting on the original motivation of creating a virtually coherent cache protocol as described in Section 1.1, the obvious next step is to apply such protocol to accelerators.

6.1 Baseline

6.1.1 Evaluation Technique

There are a wide range of workloads for different types of accelerators, and they can have distinctive memory access patterns; the effect of the virtual cache protocol will be different for each memory access pattern. At one extreme, we have stream accelerators. The memory access pattern of stream accelerators is linear and therefore the next address to access is very predictable, and it is easy to implement prefetching so that memory access and accelerator-specific computation can be performed in parallel. For these accelerators, consideration of memory bandwidth usually dominates that of memory access latency.

On the other hand, we can have accelerators that perform a lot of random accesses. Many data structures are sparse in nature, e.g. linked lists, sparse matrices, trees and graphs. If the accelerator has to work with these types of data structures (e.g. performing ray tracing using bounding volume hierarchy) or needs to perform arbitrary searches into otherwise continuous data structures (e.g. looking up from a hash table), then many of the accesses are random accesses and data-dependent. For these accesses, a shorter memory access latency can make a very significant performance improvement while increasing the bandwidth may have little effect.

Of course, many workloads are often a mixture of these two extremes; there are some sequential memory accesses but also some random accesses. For quantification purposes, I decided to use a synthetic workload that combines both types of these memory access patterns. The accelerator works on data with total size of M bytes but divided into N byte chunks. The accelerator will be supplied with a pointer for a request; it will read N bytes from the pointer and process these bytes; after processing, it will follow the pointer located at the end of these bytes for the next chunk. The last chunk contains the pointer to the output buffer, with the least significant bit tagged to 1 to indicate the end of input. Upon encountering, the accelerator will halt and write the final result to the designated buffer. Memory access to the next chunk is not allowed to overlap with the computation, mimicking the random-access pattern. By tweaking N, the workload can exhibit different memory access behaviours. If N is set to a single cache line size, then this represents a random-access workload; if N is set to M, then this represents a purely sequential workload.

For the actual processing done to these bytes, I decided to simply calculate their SHA1 hash. I selected SHA1 for two reasons. First, hashing contents in SHA1 takes an intermediate amount of cycles. If the selected task is too short (e.g. memory copy), then the latency hiding effect of a streaming access pattern would not be visible; if the selected task is too long (e.g. matrix multiplication), then the difference in memory performance would be masked by the long running time of the accelerator. A naive SHA1 accelerator takes 80 cycles to process a 64-byte block, which is a good balance between the two extremes. Second, SHA1 is a widely used hash function. Although it is no longer a cryptographically secure hash function [110], it is still widely used in many applications, e.g. git [80].

6.1.2 Baseline Accelerator

The baseline accelerator implemented for the synthetic workload is shown in Figure 6.1. This accelerator implementation is shared by most configurations evaluated in this chapter.

The accelerator is divided into three parts, a controller, a scatter-gather capable DMA engine, and a stream accelerator. The controller accepts requests via a memory-mapped I/O (MMIO) interface and translates them into instructions for the DMA engine. The DMA fetches the data from a (possibly modified) TileLink link, and feeds the data into the stream accelerator with an AXI4-Stream interface. The last input block is specially indicated to the stream accelerator; upon receiving the last block, the stream accelerator will output the finalised SHA1 value back to the DMA unit via another AXI4-Stream interface, which will in turn write the hash to a location designated by the controller. When the writeback concludes, the controller will generate a pulse to the completion notification port, which may be connected directly to the interrupt controller.

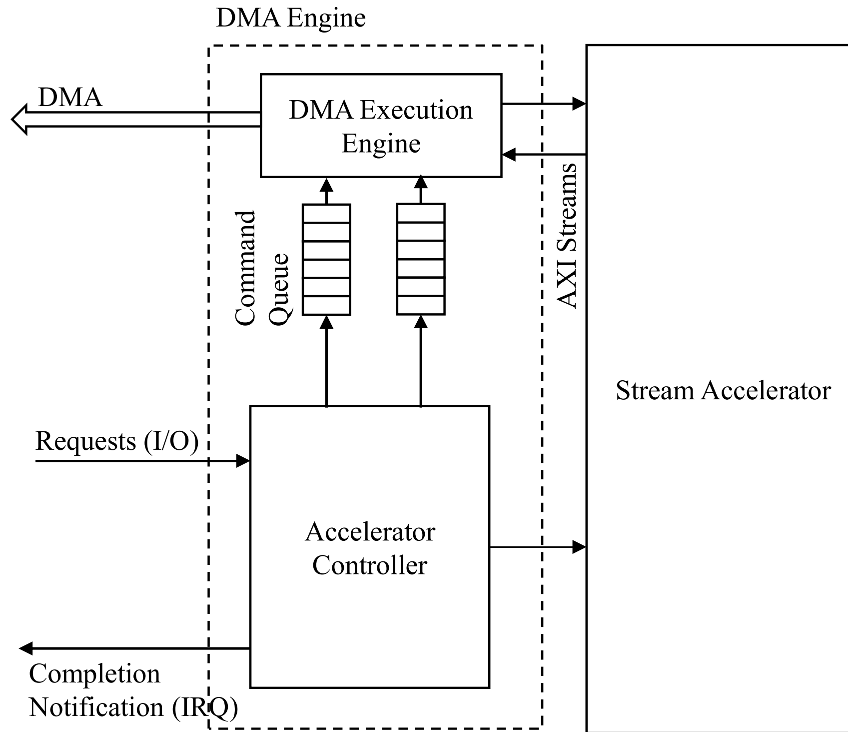


Figure 6.1 Accelerator implementation

This accelerator can be connected directly to the SoC as an I/O device; in this case all the addresses processed are physical addresses. This is used as one of the baselines. To use this accelerator, acceleration initiation is performed by a driver and userspace programs need to communicate with it via system calls. Because physical-address-based accelerators cannot access arbitrary userspace memory, data to be processed have to be copied into a dedicated memory region (similar to copying textures to GPU memory in a graphical application). When used as a baseline, I account for this extra overhead with a memory copy (together with the associated changes necessary to ensure the accelerator to walk the next chunk through pointers correctly).

For streaming only access patterns that do not require accelerator-initiated random memory accesses, it is also possible to omit the memory copy by having the driver performing the address translation of all the pages needed ahead of time and send them to the scatter-gather DMA engine. The driver I implemented supports this mode of operation. In terms of the workload described in Section 6.1.1, this mode only works if $N = M$.

6.1.3 FlexAcc

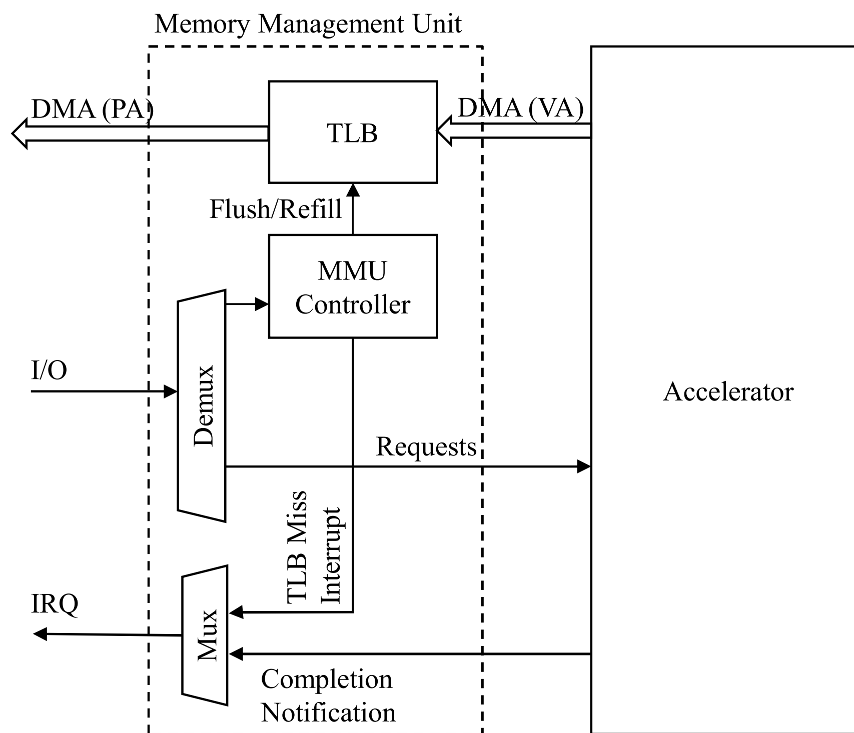


Figure 6.2 FlexAcc design

FlexAcc is the framework I built to empower accelerators to use virtual addresses. Its design is illustrated in Figure 6.2. To achieve virtual addressing capability, FlexAcc includes a memory management unit, translating virtual addresses generated by the accelerator into physical bus addresses for transactions. A FIFO TLB is provided. FlexAcc does not include a page table walker and relies on the OS to perform software TLB refill. Whenever a TLB miss occurs, the accelerator will suspend the request and trigger an interrupt. The driver is responsible for translating the virtual address to a physical address, refilling the TLB and resuming the accelerator.

A Linux driver was implemented for this framework. The driver handles device initialisation and software TLB refill. As a performance-enhancing measure, I have added prefetching to this driver implementation; for each TLB miss interrupt, the driver will insert the missing page into the TLB along with the extra 3 consecutive pages, if they are valid. The driver exposes the device as a file `/dev/flexacc0`, which can be opened by userspace programs. The userspace program can use `mmap` to access accelerator's configuration and command

MMIO registers. The interrupt signal is also exposed to the userspace program: the userspace program can use `epoll` to wait until an interrupt is triggered. Using async IO makes FlexAcc easily integrable with event-loop based programs.

FlexAcc's performance was not satisfactory. When the amount of data is small, performing SHA1 acceleration using FlexAcc can yield worse performance than computing the hash purely in software, making it a decelerator instead. The observation is consistent with literature which suggests that servicing TLB misses and page faults from peripherals like GPU is considerably slower than that from CPU [118]. The unsatisfactory performance led me to believe that the address translation should be performed near the LLC instead of the accelerator, and was a major motivation for the design of the virtual cache coherence protocol.

Although FlexAcc was not successful, I believe that it can serve a useful data point as one of the baselines. For the baseline, the TLB size is chosen to be 32 entries.

6.1.4 Summary of Baselines

For all accelerators described in this section, they are connected as peripheral devices attached to a Muntjac system with physical cache coherency, using the same setup as described in Section 5.4. I have also connected them to the virtually coherent Muntjac variant; there are no observable differences in throughput since they still communicate with the L2 using physical addresses. The accelerator is clocked at 80 MHz, the same as the clock used for the cores.

All accelerators described in this chapter will be invoked by a userspace program. The userspace program will prepare the data to be processed, send the request to the accelerator under evaluation (through the driver, e.g. for the physical case, or through MMIO, e.g. for FlexAcc) and record the time spent waiting for the accelerator to complete.

Apart from the aforementioned hardware baselines, I also ported an optimised software SHA1 implementation as another baseline.

6.2 Non-coherent Accelerators

First, I explore accelerators that are virtually addressed but not cache coherent. Non-coherent[1] here means that accelerators are not connected to the cache hierarchy with a cache coherent interconnect, so that CPU cores cannot invalidate cache lines from accelerators. Almost all

[1] In the sense that CPU memory accesses would not probe accelerators. As discussed in Section 2.2, this is not to be confused with I/O coherency. All evaluations in this chapter are performed on I/O coherent systems.

traditional hardware accelerator falls into this category – the application submits a request to the accelerator, which does some memory accesses and computation, and eventually notify the application upon completion using interrupts. No synchronisation happens between the application code and the accelerator when the task is being executed.

6.2.1 Virtual Addressing

With the proposed virtual cache protocol, the accelerator could perform DMA accesses or do random memory accesses using virtual addresses, without having to contain TLBs and logic related to address translation. However, simply connecting the accelerator to the interconnect would be insufficient, due to OS's use of page files (swaps) and memory overcommitment. When a virtual address is accessed by an application, the access can be logically valid, but the associated page may not be present in memory, e.g. the page may be stored on disk, either due to the memory being memory-mapped file or due to the page being swapped out. This can also happen for fresh and unused pages, as the OS may delay the actual allocation of a page until the page is accessed; a technique frequently used for private and anonymous `mmaps` and for stacks. Although these pages are logically accessible from the userspace code, the PTE for those pages are marked as invalid. For general-purpose cores, accessing these pages would simply trigger a page fault and the page fault handler could fix up the PTE, and the execution could continue upon return from the handler; this is however not feasible for accelerators since page fault handlers could not typically execute on the accelerators themselves.

Therefore, I have inserted a thin middle layer between the accelerator and the interconnect, as shown in Figure 6.3. An access guard is placed between the accelerator and the interconnect. When a process opens an accelerator device file for access, the driver will configure the access guard with the ASID and root page table number of that process. Normally, the guard will simply forward any memory access requests from the accelerator to the interconnect with information necessary for address translation attached, and forward responses back from the interconnect to the accelerator if accesses are successful.

If an access is denied due to a page fault however, the triggering address will be stored and an interrupt will be triggered. The driver could fetch the address and handle it similarly to a normal page fault triggered on an application core. After the PTE has been updated and the address becomes accessible, the driver will tell the middle layer to replay the failing transaction.

Logically, the page fault triggered by the accelerator is delegated to a general-purpose core. From the accelerator's perspective, the page fault in this case is invisible and transparently

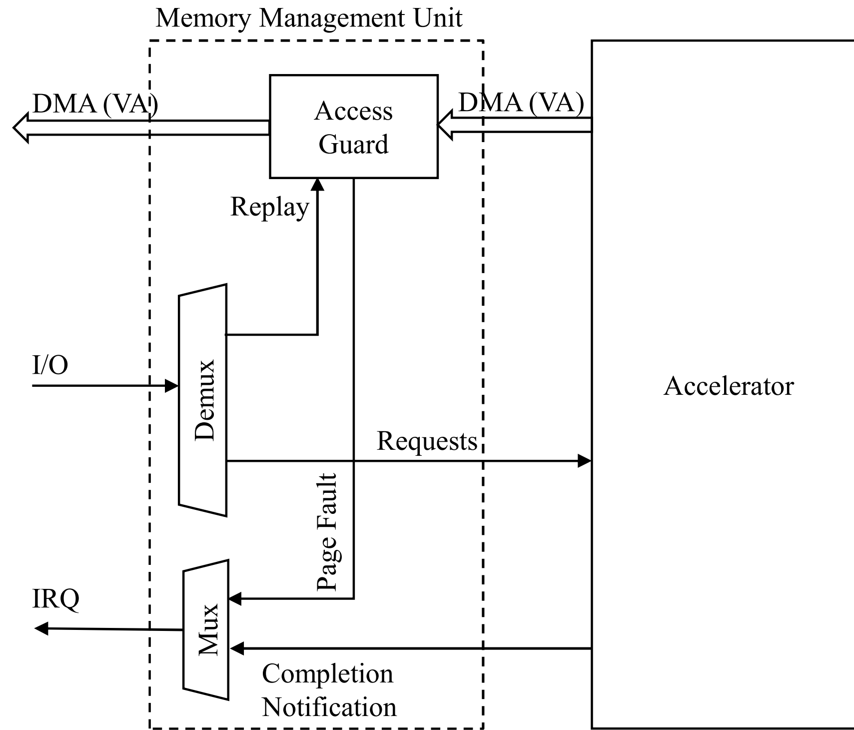


Figure 6.3 Virtually addressed accelerator framework

handled. This design also ensures that both the interconnect and the accelerator run the virtual cache coherence protocol described in Chapter 5 unmodified.

This virtual addressed accelerator is connected to the virtually coherent Muntjac setup for evaluation. It should be noted that this virtual addressing approach for accelerators does not *require* the use of a virtual cache coherence protocol, because all accesses are uncached and therefore do not have to be coherent. We can connect the virtual DMA host port in Figure 6.3 to the L2 MMU unit as described in Section 5.3.8 and Figure 5.27 and convert it to a physical DMA host port, and then used with physically coherent processors. I have also included this variant for evaluation; the TLB size used inside the MMU unit is chosen to be 32 entries. This configuration is referenced in figures and in later texts as *Hybrid addressing*.

6.2.2 Evaluation

Figure 6.4 shows the performance of the accelerator, in terms of throughput. The total data size of all chunks is fixed to be 64 KiB, but each data point indicates the performance of a particular configuration under a particular chunk size N. As expected, the software performance is very stable with different N; the variance is only around 1 Mbps. As discussed in Section 6.1.3, FlexAcc has a huge overhead for TLB misses, so when the chunk size is

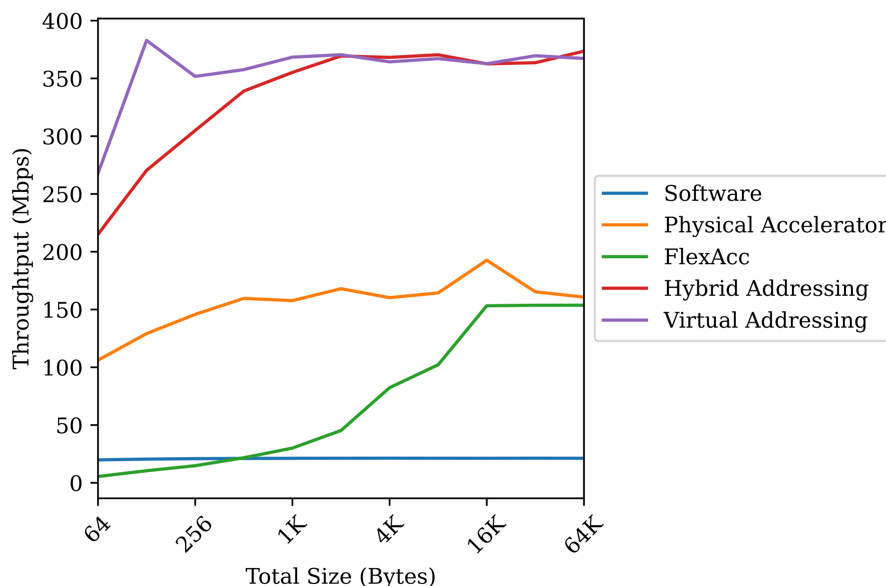


Figure 6.4 Accelerator throughput in relation to chunk size N, data size $M = 64$ KiB

small, the overhead overshadows the computation acceleration and causes the performance to be worse than its software counterpart; it only outperforms software implementation for chunk sizes larger than 512 bytes. The throughput plateaus after 16 KiB at 154 Mbps; this is caused by the prefetching behaviour where 4 pages are fetched at a time for a TLB miss. As described in Section 6.1.2, we are testing memory access patterns with random access here as $N \neq M$; to test the physical accelerator here we need to memory copy the data to a dedicated region and perform pointer fixups. Despite this additional overhead, it still performs better than FlexAcc for all chunk sizes (one reason that FlexAcc is abandoned and the virtual cache work is started in the first place). The physical accelerator reaches around 160 Mbps for large chunk sizes.

Both the virtual and hybrid addressing schemes have significant edge over the physical accelerator. The virtual addressing variant has better performance at small chunk size; the difference becomes negligible when chunk size increases. Figure 6.5 shows the same data but in terms of time spent per cache line: when $N = 64$, the virtual variant takes 1.9 μs per cache line, while the hybrid variant takes 2.4 μs per cache line. For a freshly supplied data chunk from the application, it is easy to see that the data will not reside inside the TLB of the MMU unit used in the hybrid configuration; on the other hand, because the virtual variant uses virtual cache coherence protocol and is directly attached to the virtual interconnect, all accesses go through the shared L2 and are translated to physical addresses by the shared TLB inside the L2, so it can take advantage of the entries that already reside in the shared TLB

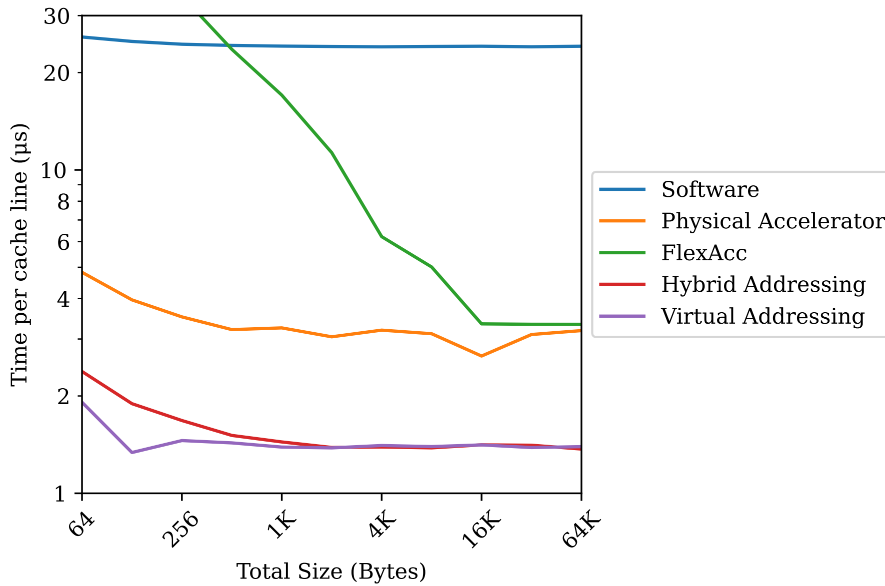


Figure 6.5 Time spent per cache line, in relation to chunk size N, data size $M = 64$ KiB

when the application core prepares the data to be sent to the accelerator. The time difference is the time that the hybrid version spent walking the page table. Because page table walking only needs to be performed once per page, this overhead is diluted when the chunk size increases as the accelerator exhibits more sequential access pattern than random access. With a large chunk size, these two versions both peak at around 368 Mbps.

The metric mentioned about tests the case where the size is 64 KiB. In practice, there are many workloads that deal with smaller data sizes. Figure 6.6 and Figure 6.7 show the performance of the same setups with different total data sizes, in streaming workload. In this test, a copyless physical accelerator that uses scatter-gathering DMA is included as mentioned in Section 6.1.2. It can be seen that there is a high fixed overhead for any type of accelerator tested; the exact cost varies between configurations, from 259 μs in virtual addressing setup to 849 μs in FlexAcc. This fixed overhead includes the ahead-of-time address translation overhead in the physical accelerator, or the on-the-demand address translation in the FlexAcc. For all accelerators, the fixed cost also includes the overhead of interrupts to signal request completion, as the thread that invokes the accelerator puts itself into sleep until an interrupt wakes it up. Switching to kernel is required for handling interrupts, and many kernel components, such as the driver for the interrupt controller and the driver for the accelerator, needs to perform work, making interrupt handling a slow operation. The overhead causes all configurations to be a decelerator rather than an accelerator when compared to the software implementation when the size is small.

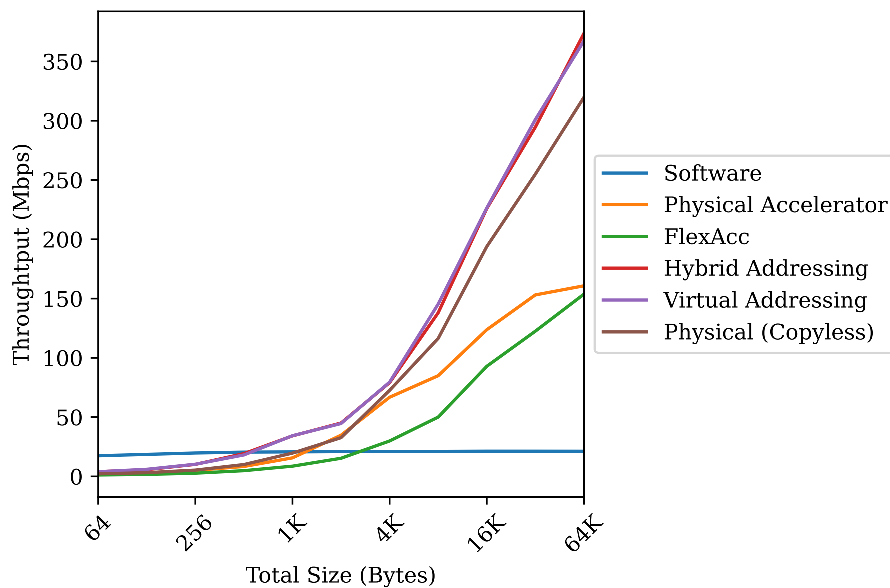


Figure 6.6 Accelerator throughput in relation to data size M for streaming workload (chunk size $N = M$)

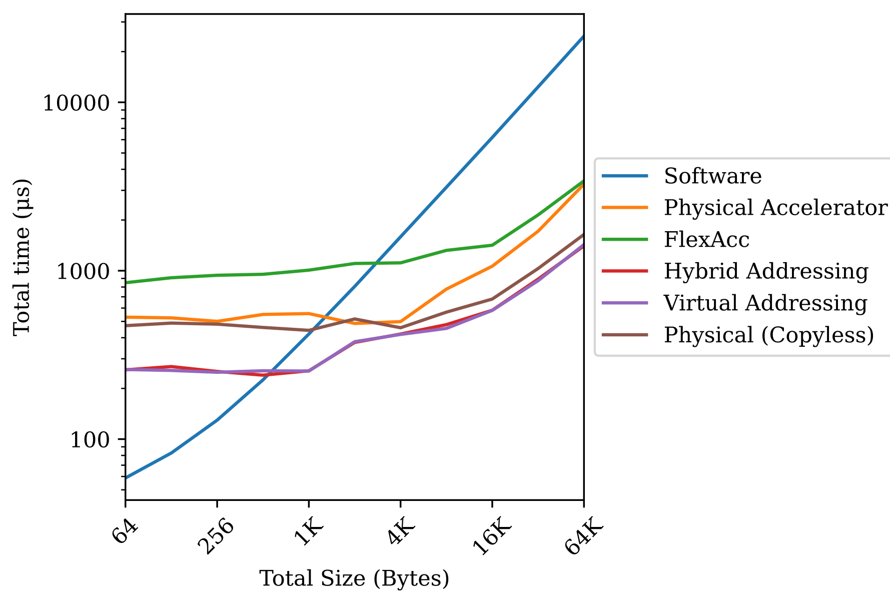


Figure 6.7 Time spent for each request, in relation to data size M for streaming workload (chunk size $N = M$)

Overall, it can be seen that accelerators utilising virtual addressing have a significant advantage for workloads that require random access. For streaming workloads, it can be seen from Figure 6.6 that it does not outperform copyless physical accelerator. However, so far only the virtual addressing part is utilised; the virtual cache coherence protocol is yet to be exploited.

6.3 Coherent Accelerators

With the virtual cache coherence protocol, it can also empower accelerators to perform coherent memory accesses to userspace addresses without incorporating any additional logic for dealing with address translation. Unlike the non-coherent accelerator, it is not possible to have an add-on address translation unit attached like the hybrid configuration discussed in the previous section.

6.3.1 Futex-based Completion Notification

In Section 6.2.1, I discussed that the use of interrupts limits the performance of accelerators when the data size is small and frequent communication with the application is needed. One solution would be to use polling instead: userspace networking libraries, such as DPDK [99] and netmap [102], use polling mechanism instead of interrupts to bypass kernel entirely for low latency networking. However, polling requires constant bus transactions and prevents application cores from idling or switching to other processes, so it is inefficient both power-wise and utilisation-wise.

Virtual cache coherency allows a different solution that requires neither polling nor interrupts. I take heavy inspiration from mutexes implemented with fast userspace mutex (futex) [41]. A syscall-facilitated synchronisation like POSIX semaphores is expensive and high latency like interrupts, while a purely userspace spinlock is cheap and low-latency but it can cause degenerate behaviours when the thread holding the spinlock is preempted [73]. Futex provides a syscall that allows threads to wait on an atomic integer accessible from userspace. The simplest futex implementation typically has two operations, `WAIT` and `WAKE`. The `WAIT` operation will atomically check that the integer still holds the expected value and put the thread into sleep, while the `WAKE` operation will resume thread(s) waiting on the specified integer.

A futex-based mutex typically uses two bits in an atomic word; one bit (`Locked` bit) to indicate whether the mutex is locked, and another bit (`Parked` bit) to indicate whether the thread is waiting for the mutex using the syscall. The operation of such mutex is illustrated in

Locked	Parked	Description	Lock Action	Unlock Action
0	0	Unlocked and no waiting threads	Set Locked to 1	N/A
0	1	Unlocked and has waiting thread(s). This is a transient state where a thread has just unlocked this mutex and waiting threads are woken up, but have not yet acquired this lock.	Set Locked to 1	N/A
1	0	Locked and no waiting thread	Either spin, yield, or: set Parked to 1, and call futex syscall with WAIT action.	Set Locked to 0
1	1	Locked and has waiting thread(s)	Same as above	Set Locked to 0, and call futex syscall with WAKE action. Set Parked to 0 if all threads have been woken up.

Table 6.1 States and actions of an example futex-based mutex implementation

Table 6.1. In uncontented operation, the atomic integer can be used like a spinlock; locking is done by setting the `Locked` bit to 1, and unlocking is done by clearing the bit to 0. These operations can proceed without any interaction with kernel space. In contended operation, the `Parked` bit will be set and futex system call with the `WAIT` operation can be used. The kernel will atomically check that the mutex is still locked and, if necessary, put the thread into sleep. When a lock with `Parked` bit set is released, the unlocking thread can use futex syscall again to wake pending threads. The key feature of futex is that the thread acquiring the lock in the userspace has full control over the locking strategy of the mutex – when contention is encountered, it can choose to continue spinning, perform spinning with exponential backoff, yield to another thread, or use futex syscall.

I have designed a mechanism that utilises the same idea for completion notification. In fact, this design requires no special system calls or new ioctls at all – it uses exactly the same futex system call as mutexes. Figure 6.8 shows the design of such an accelerator framework. Before a request starts, the userspace program is expected to allocate a word-sized integer for completion notification. This integer can be treated exactly like a mutex shown in Table 6.1. Before a request is sent to the accelerator, the `Locked` bit is expected to be set. The address of this word is provided by the userspace program to the accelerator when a request starts and is stored by the control unit. When the accelerator completes the request, the control unit will take the ownership of the cache line, set the `Locked` bit to 0. If `Parked` bit is clear, then no interrupts will be raised; otherwise, an interrupt is raised and the driver will read out the address from the control unit and perform a futex `WAKE` operation. Essentially, this

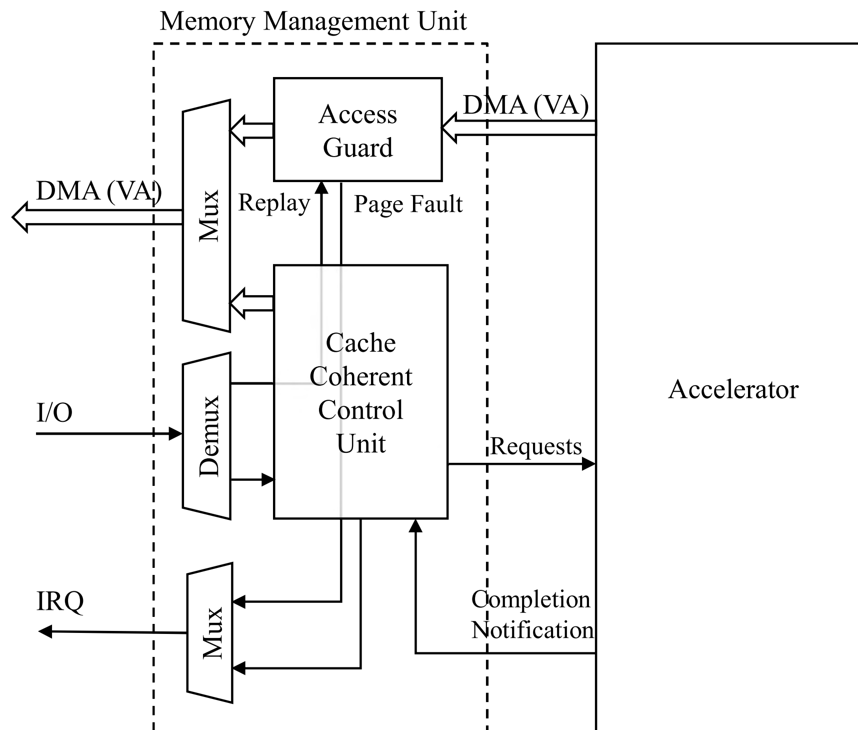


Figure 6.8 Virtually coherent accelerator framework

word-sized integer is a mutex that is to be held when the accelerator is in use, and waiting for completion is simply re-acquiring this mutex. This approach only works if the accelerator is able to perform cache coherency access due to its use of atomics.

I tested this mechanism with two configurations; one configuration simply spins on this atomic variable in userspace to wait for accelerator task completion. This is shown in the figures as `Futex (Spin)` (despite that it actually makes no futex system calls). For another configuration, it uses an adaptive strategy to determine whether it spins on the variable or whether futex syscall should be made. I have ported the strategy used when locking a mutex from a popular Rust locking crate `parking_lot` [37]. The strategy is:

- Try to set the `Locked` bit to 1 if it is currently 0.
- Relax the CPU by executing a spin loop hint instruction. This corresponds to the `pause` instruction on x86/64 and RISC-V. After that, retry setting the `Locked` bit. If it still fails, execute the spin loop hint more times per try, exponentially.
- If the `Locked` bit is still 1 after some spin attempts, then yield the thread with `sched_yield` and retry after being rescheduled by the OS.

- If the `Locked` bit is still 1 after some yield attempts, then proceed and park the thread using futex syscall with `WAIT` action.

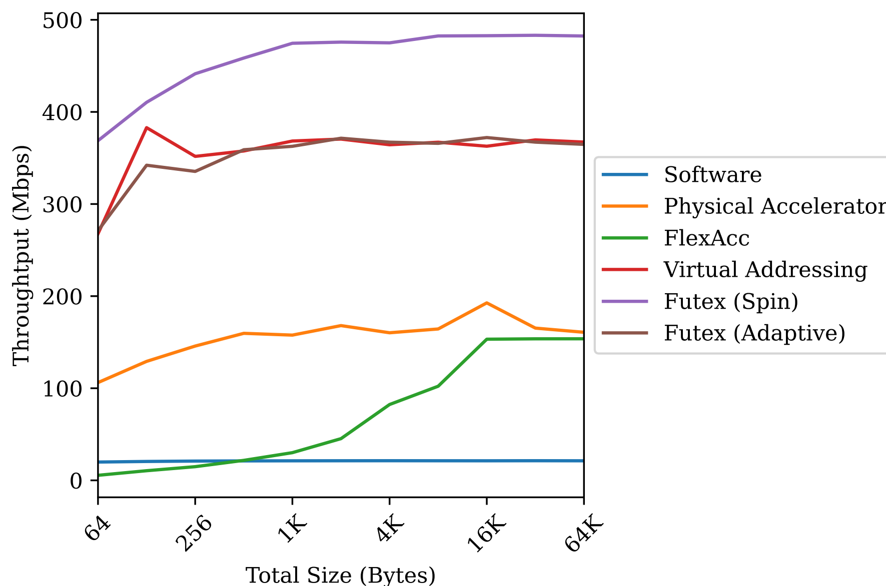


Figure 6.9 Accelerator throughput of futex-based completion notification, in relation to chunk size N, data size $M = 64$ KiB

Figure 6.9 shows the performance of the virtually addressed accelerator after adopting futex-based completion notification with $M = 64$ KiB data size. Spinning on the atomic variable is the fastest approach; it cuts down the overhead significantly as it completely bypasses the kernel. Because of the total data size, in this scenario the adaptive variant has always saturated the quota used for spinning and yielding, so it resorts to the futex syscall; in this situation it has almost identical performance to the virtually addressed accelerator that uses `ioctl` to pass the completion interrupt to the userspace.

The data is more interesting when we look at the streaming workload with variable data sizes, as shown in Figure 6.10. The spinning configuration is undoubtedly the fastest again, significantly better than any other configuration tested. The adaptive variant is very different from the virtually addressed accelerator in this scenario though. It is divided into 3 increasing segments. These are actually a reflection of the adaptive strategy mentioned above. The first segment corresponds to the spinning and relaxing step, the second segment corresponds to the yielding phase, while the third segment overlaps with the virtual variant, indicating that futex syscall is being used.

Although slower than the spinning variant, the adaptive approach is still, in all circumstances, faster or has similar performance to ones without this futex-based completion

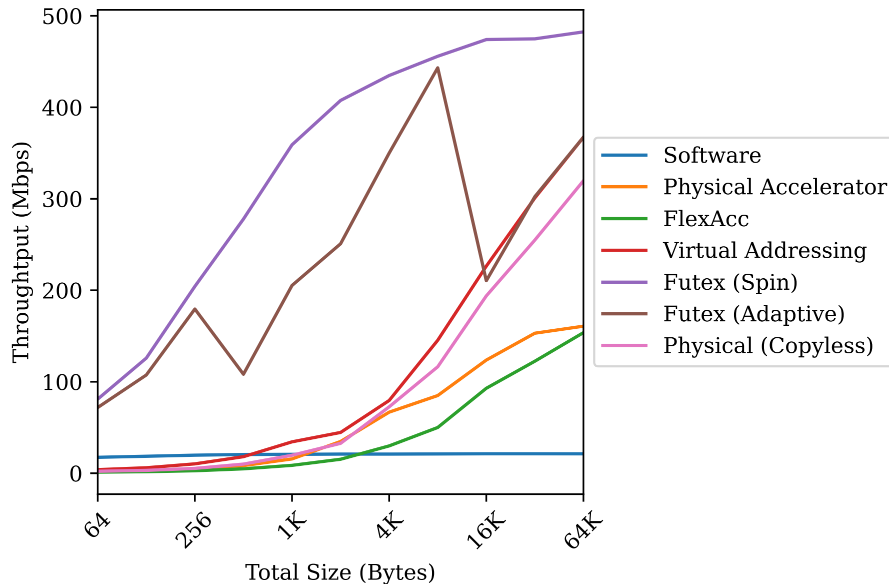


Figure 6.10 Accelerator throughput of futex-based completion notification, in relation to data size M for streaming workload (chunk size $N = M$)

notification mechanism. Furthermore, it can be superior to the spinning variant in complex, loaded environments; spinning on the atomic variable has all the disadvantages of a spinlock that is present in real life but not in a clean, unnoisy test environment. The futex-based notification mechanism provides flexibility to the applications; not only that they can decide whether they want to spin for low latency or sleep for low CPU utilisation, this decision can be made after the request has been sent to the accelerator.

Combining the virtual addressing used by the accelerator's memory access and the cache coherency which empowers the futex-based completion notification scheme, we provide a way for userspace programs to communicate and synchronise with accelerators with little to no interaction with the OS kernel. Bypassing the kernel provides significant speedup while the use of virtual addressing allows the accelerator to perform any kind of memory accesses, not having to be restricted to a particular memory pattern like streaming or striding. Use of the virtual cache coherence protocol means that all of these pros can be achieved cheaply: the cache coherency control unit in Figure 6.8 only costs 527 lookup tables (LUTs) and 715 registers targeting Xilinx Kintex 7, a very small footprint that is barely larger than a TileLink FIFO unit.

6.3.2 Using a Core as an Accelerator

Some previous works, like conservation cores [117], explored turning entire program routines into accelerators or power-saving cores, but only do so for bare-metal applications and do not cover virtual addressing and time-sharing OSes. The framework described above and the virtual cache coherence protocol effectively permit any userspace programs to be turned into accelerators regardless of how unpredictable memory access patterns are.

In fact, it is even possible to run the entire userspace application on such an "accelerator" unmodified. As a proof of concept, I customised Muntjac and built a special version of it. It has all privilege levels other than the user level removed, along with all system instructions and CSRs that come with these privilege levels. Instead of a `SATP` register accessible from the pipeline itself, it provides MMIO registers that only drivers can use. Despite being a RISC-V core, it is not visible to the core part of the kernel; the driver exposes it as a device file like accelerators do. When a process opens the device file, the driver will use that process's `SATP` register to configure the accelerator. The userspace program can then send a function pointer to be executed on this accelerator core, just like how programs send requests to other accelerators. Whenever a trap happens, e.g. page faults or system calls, the customised core will deliver it to the driver on a general-purpose core, and the driver can fix the page table (similar to the access guard mentioned in Section 6.2.1) or proxy the system call.

As an example, I have implemented a SHA1 extension to this core. The customised core uses the configuration where FPU registers are enabled and have FPU replaced by the custom accelerator unit. The implemented instruction set extension reuses these registers as states for computation. A single instruction is added, which uses `fa0-fa7` (64 bytes in total) as data input and `ft0-ft4` (only uses least significant 32-bits, 20 bytes in total) as hash state input/output. There are no built-in DMA units associated with the SHA1 unit; all memory accesses are performed by the processor pipeline, through the VIVT data cache that talks the proposed virtual cache coherence protocol.

Because the accelerator-integrated core still retains the capability to run all user-level RISC-V instructions, such design offers extreme flexibility to the program. The program can decide the way that the accelerator core and the rest of the program communicate. If desirable, the accelerator core can run data preparation work on itself, as long as it is not I/O heavy (since system calls are not handled on the accelerator core itself but delegated back to general-purpose cores). In fact, I have designed an API in Rust that even allows the program to use the accelerator transparently and safely, as shown in the Listing 6.1.

I was able to run the same benchmark used for all other accelerators with this API. The result is shown in Figure 6.11 (spinning synchronisation primitives were used to communicate between the accelerator core and general-purpose cores). Although not as performant

```rust
/// This is a token type. Code that possesses this token is running on
/// the accelerator and therefore can use accelerator-specific custom
/// instructions without triggering illegal instruction traps.
struct OnAccelerator { ... };

impl !Send for OnAccelerator {}
impl !Sync for OnAccelerator {}

fn digest(
    token: &OnAccelerator, state: &mut [u32; 5], block: &[[u8; 64]]
);

struct Accelerator { ... };

impl Accelerator {
    pub fn open() -> Result<Accelerator>;

    pub fn run<R, F>(&mut self, f: F) -> R
        where F: FnOnce(&OnAccelerator) -> R;
}

// Example code

let acc = Accelerator::open()?;

acc.run(|token| {
    // Code here is running on the accelerator core.
    // Still is capable to anything, just like a normal thread, but
    // system calls are slightly slower as they are delegated back to
    // general-purpose cores. E.g, prepare the data to hash.
    let data = ...;

    let mut state = [
        0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476, 0xc3d2e1f0
    ];
    digest(token, &mut state, data);

    // Could do some postprocessing on the hash.
});
```

Listing 6.1 API and example usage of the accelerator core

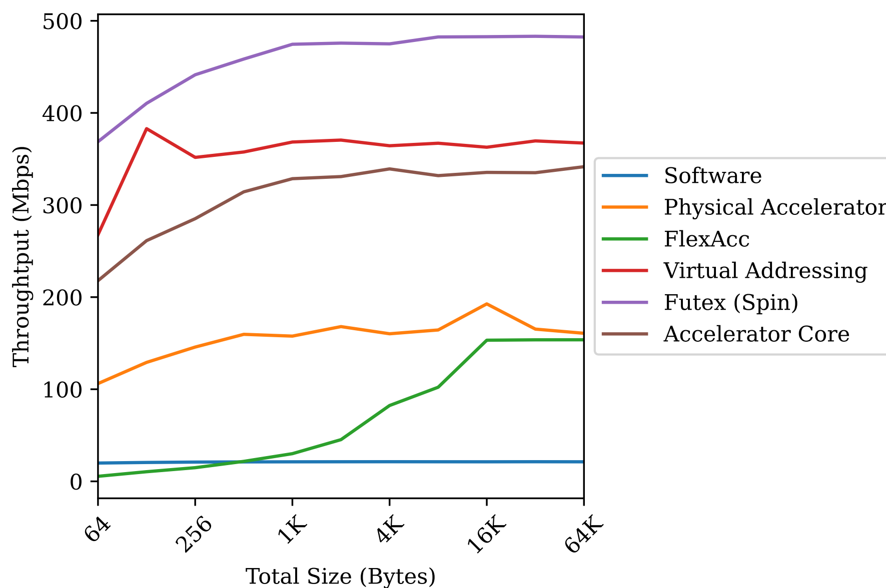


Figure 6.11 Accelerator throughput of core with integrated accelerator, in relation to chunk size N, data size $M = 64$ KiB

as dedicated accelerators with DMAs, it demonstrates its potential by achieving decent performance while being extremely flexible in how it may be used in the software.

In conclusion, the ability to be cache coherent *and* use virtual addressing from an accelerator or a peripheral device unlocks a lot of possibilities for userspace programs. It allows efficient communication between userspace programs and devices with minimal kernel involvement, and can do so without invasive programming model changes and code changes. The reduced latency also permits a greater fraction of workloads to be accelerated, by making small pieces of work profitable to accelerate. There are some further extensions possible that will be discussed in Section 7.1.

Chapter 7

Conclusion and Future Work

This dissertation presents the design of a practical virtual cache coherence protocol. It also contributes in the areas of fast simulation techniques and provides a complete hardware implementation of a multi-core processor with virtual caching.

On the simulation front, I presented techniques that can be used to analyse TLB behaviour on top of existing simulators like QEMU, and provided analysis of feasibility and advantage of exploiting shared TLBs by using ASID, and the process of which influences the software requirement in the RISC-V specification that non-zero ASIDs must have identical meanings on all cores. I presented techniques that allow combination of DBT and cycle-level simulation. The simulator built, R2VM, achieves $\sim$30 MIPS in cycle-level simulation in lockstep execution mode, providing a significant speedup compared to existing cycle-level simulators in cycle-level mode. In a multi-threaded non-cycle-level execution mode, it can achieve >400 MIPS per thread, outperforming QEMU's multi-threaded TCG.

I designed, implemented and open-sourced, with collaboration with Dr Daniel Bates in regard with testing, Muntjac, a collection of IPs that include processor pipeline, cache, and cache coherency components. Muntjac core can achieve Dhrystone score of 2.17 DMIPS/MHz and CoreMark score of 3.01 CoreMark/MHz. Muntjac has been a good baseline for my other experiments, and I believed the modularly designed Muntjac can be a good starting point for other applications as well.

Finally, an important part of this thesis is the design of a virtual cache coherence protocol, along with the hardware implementation of cache and interconnect components communicating in this protocol. Getting the hardware implementation functional was challenging due to its cross-stack nature; an issue could only be fully understood when the OS, the interconnect and the coherence protocols were all considered, and similar to many other concurrency-related issues, debugging was hard as the cause was not deterministic and usually would only happen billions of cycles after booting. The challenges however were

overcome and a working protocol and implementation is presented; the final hardware is able to boot Debian on it with a modified Linux kernel, and can run unmodified RISC-V programs. Reflecting on the hypothesis in Section 1.1 and the design principles in Section 4.1, I believe that the intention was met. Apart from using it to implement VIVT CPU L1 caches, I implemented and evaluated different accelerators that utilises virtual addressing, and implemented a completion notification that makes better use of the virtual cache coherence protocol, and demonstrated its flexibility by attaching a whole RISC-V core as a peripheral device.

7.1 Future Work

There are a multitude of future works possible in each of the areas discussed above.

7.1.1 Virtualisation-based Memory Access Acceleration in Full-system Binary Translation

For simulator/binary translator, both R2VM and QEMU uses a software direct-mapped TLB and generate code that checks this TLB for each memory-access instruction. This software TLB lookup requires at least two extra memory accesses for each original memory-access instruction and therefore hurts performance.

I hypothesised that it is possible to use virtualisation to speedup this process, by making the simulator run as a hypervisor, e.g. using the kernel-based virtual machine (KVM) APIs. The code generated by DBT can be executed as a guest supervisor (virtual machine), and CPU's second-level address translation mechanism can be used to do the address translation instead of the software TLB lookup code. Instead of maintaining a software TLB structure, the simulator can instead maintain a shadow page table. This technique should reduce number of extra memory accesses to one or even none, thus providing considerable performance improvement.

7.1.2 Just-in-time Compilation of Advanced Pipeline Models

Currently, R2VM only implements an in-order pipeline model. While it is possible to hook in more complex models, at the current stage doing so would require custom generation of assembly code that updates the state. It is possible to describe pipeline models in code or DSL and then generate assembly code automatically. Doing so may require usage of more powerful just-in-time (JIT) compilation libraries, like LLVM's JIT APIs [76] or Cranelift [1].

7.1.3 Extensions to Muntjac

Muntjac provides a basic design that is suitable for a baseline for education or evaluation purposes. There are lots of possible extensions possible to it to improve performance, e.g. multiple issue, more powerful branch predictor, etc.

7.1.4 Multi-level Virtual Cache

The proposed virtual cache coherence protocol, as implemented and evaluated, supports only virtual L1 cache, with the virtual addressing terminated by a MMU in between the L1 and the L2. It does not currently support multi-level virtual caches, e.g. both virtual L1 and L2 caches with only L3 becoming physically indexed. In Section 4.6 I evaluated two solutions to the issue where dirty cache lines cannot have their virtual address translation when ASID changes: flush L1 caches on ASID changes or introducing a new level of indirection. A more complex solution would be to do both, creating a root page table number table for slow paths in case ASID changes, while keep sending current `SATP` CSR values if ASID matches. Implementing such a scheme would make it possible to have a virtually coherent L1 cache *and* a virtual coherent L2 cache, and only terminating virtual addressing and converting to physical addressing for the L3 cache, a further extension possibility.

7.1.5 Distributed Shared Caches with the Virtual Cache Protocol

In this thesis, I evaluated the proposed virtual cache protocol exclusively using a monolithic shared TLB and a single shared LLC. Many modern multiprocessor systems, however, use distributed LLCs, instead of centralised ones, and cache access delays are often non-uniform [8]. For example, Intel Skylake uses 4 slices of L3 cache connected with cores using a ring interconnect [39]. AMD's Zen 2 microarchitecture also divides L3 caches within a core complex (CCX) into 4 slices [112]. ARM's Neoverse spreads the LLC between tiles and connects them via a mesh interconnect [95].

The protocol is not dependent on the use of a centralised LLC. For example, the "cache banks" shown in Figure 5.19 could be banked, or could be tiled/distributed. Because the L2 cache in the proposed protocol is physically indexed, there is no limitation at the protocol level that precludes distributed cache designs, although the cache implementation would have be modified to include additional virtual address information on the links that interconnect cache slices, and the cache would have to include additional metadata required by the protocol as shown in Figure 4.18 and follow the state transition rules described in Section 4.8. The detailed designs and implementations of such caches are considered as possible future work.

Just scaling the caches in a design with many processors and/or accelerators is insufficient, given that the MMU needs to be accessed before accessing the cache. However, as described in the footnote in Section 4.2, the MMU, as implemented, can be duplicated or distributed, as the MMU does not store special information: the page translations stored in its TLB is not coherent and the information is available from the next-level cache. In fact, duplication of MMUs and associated TLBs are used to quantify the performance gain of inter-core TLB

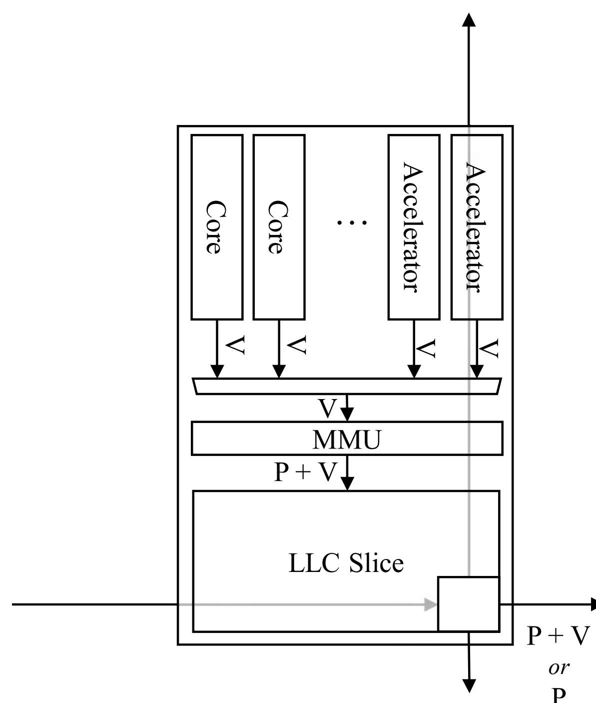


Figure 7.1 Possible system design with distributed LLC using virtual cache protocol

An example of potential designs is shown in Figure 7.1, where each tile contains its own MMU shared between host devices within the tile, and a slice of globally shared cache which connects with other tiles in a mesh network. Communications between slices primarily use the physical address (as they are physically indexed), but the virtual address could be included in messages for L1 invalidation purposes. If all cache slice stores the virtual address information of home L1 caches in their own directory, then virtual addresses do not need to be included in the messages, and the communication can be purely physical. Another possible design option is to use the virtual protocol between L1 and L2 caches only, and have a purely physical distributed LLC.

7.1.6 Multi-user and Time-shared Accelerator

For the virtually addressed accelerator or peripheral device, the current version implemented is single user only, i.e. only one process can open a device at a time. It is possible to extend it to support multiple processes. Instead of the complex single-root I/O virtualisation (SR-IOV) used in PCIe, doing so with the proposed virtual cache coherency should simply require storing multiple `SATP` values and use the correct one for each bus transactions. Even timing sharing between multiple processes should be possible.

7.1.7 Many-core Systems with Virtual Cache Coherency

I presented a multi-core (4-core) system that uses virtual cache coherency in Section 5.4. It should be possible to expand the system to many-core systems.

Such systems would likely need a more advanced L2 cache; instead of a monolithic L2 cache, for a many-core system it is likely that tiled L2 cache with mesh interconnect is required. The protocol itself does not preclude such hierarchy but obviously more engineering effort is required to implement it. The L2 cache in this thesis is inclusive for simplicity, and for a many-core system it is possible that an exclusive or non-inclusive non-exclusive policy would be beneficial, with directory metadata stored separately.

7.1.8 Seas of Accelerators with Virtual Cache Coherency

In Section 6.3, I presented a simple heterogeneous system with 4 cores and 1 cache-coherent accelerator. It should be possible to scale the number of virtually-addressed cache-coherent accelerators, as the cost of adding a new virtually coherent host is relatively low.

As accelerators can, in a low cost manner, access virtual address spaces directly and synchronise with cache coherency, it allows arbitrary program routines into accelerators, even ones that use atomics or perform memory accesses with unpredictable patterns. The low communication overhead enabled by the protocol also permit some traditionally not acceleration-worthy tasks to be converted to accelerators, e.g. even at the granularity of a single function or loop. Such features allow the limitation from the Amdahl's law [3] to be overcome by allowing a larger fraction of the program to be accelerated.

Apart from improved performance, the techniques could also be used to save power by offloading more tasks from general-purpose cores to more energy-efficient specialised hardware.

References

[1] Bytecode Alliance. Cranelift code generator. https://github.com/bytecodealliance/wasmtime/tree/main/cranelift, 2020. Accessed: 2021-11-22.

[2] Oscar Almer, Igor Böhm, Tobias Edler Von Koch, Björn Franke, Stephen Kyle, Volker Seeker, Christopher Thompson, and Nigel Topham. Scalable multi-core simulation using parallel dynamic binary translation. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 190–199, 2011.

[3] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Spring Joint Computer Conference*, pages 483–485, 1967.

[4] ARM. Cortex-A9 technical reference manual r4p1. https://developer.arm.com/documentation/ddi0388/i, 2012. Accessed: 2022-12-15.

[5] ARM. AMBA AXI and ACE Protocol Specification. https://developer.arm.com/documentation/ihi0022/hc, 2021. Accessed: 2021-09-22.

[6] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The Rocket Chip Generator. Technical report, EECS Department, University of California, Berkeley, 2016.

[7] 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 *49th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1216–1225, 2012.

[8] Rajeev Balasubramonian, Norman P Jouppi, and Naveen Muralimanohar. *Multi-core cache hierarchies*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[9] Jonathan Balkind, Katie Lim, Fei Gao, Jinzheng Tu, David Wentzlaff, Michael Schaffner, Florian Zaruba, and Luca Benini. OpenPiton + Ariane: The first open-source, SMP Linux-booting RISC-V system scaling from one to many cores. In *Third Workshop on Computer Architecture Research with RISC-V*, pages 1–6, 2019.

[10] Thomas W Barr, Alan L Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *ACM/IEEE 37th International Symposium on Computer Architecture (ISCA)*, 2010.

[11] Arkaprava Basu, Mark D Hill, and Michael M Swift. Reducing memory reference energy with opportunistic virtual caching. In *ACM/IEEE 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 297–308, 2012.

[12] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. Efficient virtual memory for big memory servers. In *ACM/IEEE 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 237–248, 2013.

[13] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[14] Thomas B Berg. Maintaining I/O data coherence in embedded multicore systems. *IEEE Micro*, 29(3):10–19, 2009.

[15] Abhishek Bhattacharjee and Margaret Martonosi. Inter-core cooperative TLB for chip multiprocessors. In *15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 359–370, 2010.

[16] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level TLBs for chip multiprocessors. In *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 62–63, 2011.

[17] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, 2008.

[18] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[19] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[20] Igor Böhm, Björn Franke, and Nigel Topham. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 1–10, 2010.

[21] Mark Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[22] Terry L. Borden, James P. Hennessy, and James W. Rymarczyk. Multiple operating systems on one processor complex. *IBM Systems Journal*, 28(1):104–123, 1989.

[23] Hadi Brais, Rajshekar Kalayappan, and Preeti Ranjan Panda. A survey of cache simulators. *ACM Computing Surveys (CSUR)*, 53(1):1–32, 2020.

[24] Michel Cekleov and Michel Dubois. Virtual-address caches. part 1: problems and solutions in uniprocessors. *IEEE Micro*, 17(5):64–71, 1997.

[25] Michel Cekleov and Michel Dubois. Virtual-address caches. 2. multiprocessor issues. *IEEE Micro*, 17(6):69–74, 1997.

[26] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A Patterson, and Krste Asanovic. BOOMv2: an open-source out-of-order RISC-V core. In *First Workshop on Computer Architecture Research with RISC-V*, 2017.

[27] Jeffrey S Chase, Henry M Levy, Michael J Feeley, and Edward D Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems (TOCS)*, 12(4):271–307, 1994.

[28] Yu-Ting Chen, Jason Cong, Mohammad Ali Ghodrat, Muhuan Huang, Chunyue Liu, Bingjun Xiao, and Yi Zou. Accelerator-rich CMPs: From concept to real hardware. In *IEEE 31st International Conference on Computer Design (ICCD)*, pages 169–176, 2013.

[29] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Architecture support for accelerator-rich CMPs. In *49th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 843–849, 2012.

[30] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. Accelerator-rich architectures: Opportunities and progresses. In *51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014.

[31] CCIX Consortium. Cache coherent interconnect for accelerators (CCIX). http://www.ccixconsortium.com, 2017. Accessed: 2019-04-12.

[32] The Embedded Microprocessor Benchmark Consortium. Coremark. https://www.eembc.org/coremark/, 2020. Accessed: 2020-04-14.

[33] Intel Corporation. Intel 80386 programmer's reference manual, 1986.

[34] The Standard Performance Evaluation Corporation. SPEC CPU® 2017. https://www.spec.org/cpu2017/, 2017. Accessed: 2020-04-14.

[35] Emilio G Cota and Luca P Carloni. Cross-ISA machine instrumentation using fast and scalable dynamic binary translation. In *15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 74–87, 2019.

[36] Crucial. Crucial 16GB DDR5-4800 UDIMM. https://uk.crucial.com/memory/ddr5/ct16g48c40u5, 2021. Accessed: 2022-08-31.

[37] Amanieu d'Antras. parking_lot. https://github.com/Amanieu/parking_lot, 2018. Accessed: 2022-08-10.

[38] Solar Designer. Getting around non-executable stack (and fix). https://seclists.org/bugtraq/1997/Aug/63, 1997. Accessed: 2022-04-12.

[39] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-generation Intel core: New microarchitecture code-named Skylake. *IEEE Micro*, 37(2):52–62, 2017.

[40] Albert Esteve, Alberto Ros, Maria E Gomez, Antonio Robles, and José Duato. Efficient TLB-based detection of private pages in chip multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):748–761, 2015.

[41] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Ottawa Linux Symposium*, page 479, 2002.

[42] Neel Gala, Arjun Menon, Rahul Bodduna, GS Madhusudan, and V Kamakoti. Shakti processors: An open-source hardware initiative. In *29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, pages 7–8, 2016.

[43] Davide Giri, Paolo Mantovani, and Luca P Carloni. NoC-based support of heterogeneous cache-coherence models for accelerators. In *12th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8, 2018.

[44] James R Goodman. Coherency for multiprocessor virtual address caches. In *2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 72–81, 1987.

[45] Google. RISCV-DV: open-source instruction generator for RISC-V processor verification. https://github.com/google/riscv-dv, 2021. Accessed: 2021-11-08.

[46] Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[47] Xuan Guo. Dynamic Binary Translator for RISC-V. https://garyguo.net/uploads/riscv-dbt.pdf, 2018.

[48] Xuan Guo. Fast TLB simulator for RISC-V systems. https://github.com/nbdd0121/TLBSim, 2019. Accessed: 2022-12-08.

[49] Xuan Guo. R2VM – Rust for RISC-V Virtual Machine. https://github.com/nbdd0121/r2vm, 2020. Accessed: 2022-12-08.

[50] Xuan Guo. InterCoreBench: Intercore performance benchmark. https://github.com/nbdd0121/InterCoreBench, 2021. Accessed: 2021-04-05.

[51] Xuan Guo and Daniel Bates. Muntjac RISC-V core. https://github.com/lowRISC/muntjac, 2022. Accessed: 2022-04-23.

[52] Xuan Guo and Robert Mullins. Fast TLB Simulation for RISC-V Systems. In *Third Workshop on Computer Architecture Research with RISC-V*, 2019.

[53] Xuan Guo and Robert Mullins. Accelerate Cycle-Level Full-System Simulation of Multi-Core RISC-V Systems with Binary Translation. In *Fourth Workshop on Computer Architecture Research with RISC-V*, 2020.

[54] Xuan Guo, Daniel Bates, Robert Mullins, and Alex Bradbury. Muntjac – Open Source Multicore RV64 Linux-capable SoC. In *First Workshop on Open-Source Computer Architecture Research*, 2022.

[55] Xuan Guo, Daniel Bates, Robert Mullins, and Alex Bradbury. Muntjac multicore RV64 processor: introduction and microarchitectural guide. Technical report, University of Cambridge, 2022.

[56] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. Rebooting virtual memory with midgard. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 512–525, 2021.

[57] Nastaran Hajinazar, Pratyush Patel, Minesh Patel, Konstantinos Kanellopoulos, Saugata Ghose, Rachata Ausavarungnirun, Geraldo Francisco de Oliveira Jr, Jonathan Appavoo, Vivek Seshadri, and Onur Mutlu. The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[58] Per Hammarlund, Alberto J Martinez, Atiq A Bajwa, David L Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, et al. Haswell: The fourth-generation Intel core processor. *IEEE Micro*, 34(2):6–20, 2014.

[59] Yuchen Hao, Zhenman Fang, Glenn Reinman, and Jason Cong. Supporting address translation for accelerator-centric architectures. In *IEEE 23rd International Symposium on High Performance Computer Architecture (HPCA)*, pages 37–48, 2017.

[60] Swapnil Haria, Mark D Hill, and Michael M Swift. Devirtualizing memory in heterogeneous systems. In *23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 637–650, 2018.

[61] John Hauser. Berkeley TestFloat. http://www.jhauser.us/arithmetic/TestFloat.html, 2018. Accessed: 2021-04-10.

[62] Kim Hazelwood. *Dynamic binary modification: Tools, techniques, and applications*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[63] Mark D Hill, Susan J Eggers, James R Larus, George S Taylor, Glenn D Adams, Bidyut K Bose, Garth A Gibson, Paul M Hansen, John Keller, Shing I Kong, et al. *SPUR: a VLSI multiprocessor workstation*. University of California, 1985.

[64] Apple Inc. Automatic Reference Counting – The Swift Programming Language (Swift 5.7). https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html, 2022. Accessed: 2022-09-17.

[65] Bluespec Inc. Bluespec, Inc. to Open Source Its Proven BSV High-level HDL Tools. https://bluespec.com/2020/09/06/bluespec-inc-to-open-source-its-proven-bsv-high-level-hdl-tools/, 2020. Accessed: 2020-09-28.

[66] Intel Inc. System programming guide. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Volume 3, 2019.

[67] Bruce Jacob. Segmented addressing solves the virtual cache synonym problem. Technical report, University of Maryland, 1997.

[68] Bruce Jacob and Trevor Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 18(4):60–75, 1998.

[69] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[70] Stefanos Kaxiras and Alberto Ros. A new perspective for efficient virtual-cache coherence. In *ACM/IEEE 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 535–546, 2013.

[71] The kernel development community. Kernel samepage merging. https://docs.kernel.org/admin-guide/mm/ksm.html, 2009. Accessed: 2022-12-10.

[72] Jesung Kim, Sang Lyul Min, Sanghoon Jeon, Byoungchu Ahn, Deog-Kyoon Jeong, and Chong Sang Kim. U-cache: a cost-effective solution to synonym problem. In *IEEE 1st International Symposium on High Performance Computer Architecture (HPCA)*, pages 243–252, 1995.

[73] Aleksey Kladov. Spinlocks considered harmful. https://matklad.github.io/2020/01/02/spinlocks-considered-harmful.html, 2020. Accessed: 2021-03-29.

[74] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (SP)*, pages 1–019, 2019.

[75] Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula. Fusion: design tradeoffs in coherent cache hierarchies for accelerators. *ACM/IEEE 42th Annual International Symposium on Computer Architecture (ISCA)*, pages 733–745, 2015.

[76] Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, 2008.

[77] John Levesque, Jeff Larkin, Martyn Foster, Joe Glenski, Garry Geissler, Stephen Whalen, Brian Waldecker, Jonathan Carter, David Skinner, Helen He, et al. Understanding and mitigating multicore performance issues on the AMD Opteron architecture. Technical report, Berkeley Lab Scientific Publications, 2007.

[78] Ankur Limaye and Tosiron Adegbija. A workload characterization of the SPEC CPU2017 benchmark suite. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 149–158, 2018.

[79] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: reading kernel memory from user space. In *27th USENIX Conference on Security Symposium*, 2018.

[80] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc., 2012.

[81] lowRISC. lowRISC chip. https://github.com/lowRISC/lowrisc-chip, 2019. Accessed: 2020-03-02.

[82] lowRISC. Ibex RISC-V core. https://github.com/lowRISC/ibex, 2021. Accessed: 2021-09-24.

[83] lowRISC. Open source silicon root of trust (RoT) | OpenTitan. https://opentitan.org/, 2021. Accessed: 2021-09-24.

[84] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(1):2, 2013.

[85] Hugh McIntyre, Srikanth Arekapudi, Eric Busta, Timothy Fischer, Michael Golden, Aaron Horiuchi, Tom Meneghini, Samuel Naffziger, and James Vinh. Design of the two-core x86-64 AMD "Bulldozer" module in 32 nm SOI CMOS. *IEEE Journal of Solid-State Circuits*, 47(1):164–176, 2011.

[86] Seung Won Min, Sitao Huang, Mohamed El-Hadedy, Jinjun Xiong, Deming Chen, and Wen-mei Hwu. Analysis and optimization of I/O cache coherency strategies for SoC-FPGA device. In *29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 301–306, 2019.

[87] Inc MIPS Technologies. MIPS R10000 Microprocessor User's Manual Version 2.0, 1996.

[88] Sparsh Mittal. A survey of techniques for architecting TLBs. *Concurrency and Computation: Practice and Experience*, 29(10):e4061, 2017.

[89] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 89–100, 2007.

[90] Rishiyur Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 69–70, 2004.

[91] Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. Efficient synonym filtering and scalable delayed translation for hybrid virtual caching. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 217–229, 2016.

[92] Anup Patel. [PATCH v5] RISC-V: Implement ASID allocator. https://lkml.org/lkml/2021/2/3/224, 2021.

[93] Igor Pavlov. 7-Zip LZMA Benchmark. https://www.7-cpu.com/, 2019. Accessed: 2020-04-14.

[94] PCI-SIG. PCI Express Base Specification Revision 3.1a. https://pcisig.com/specifications, 2015. Accessed: 2019-02-20.

[95] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, et al. The Arm Neoverse N1 platform: Building blocks for the next-gen cloud-to-edge infrastructure SoC. *IEEE Micro*, 40(2):53–62, 2020.

[96] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, et al. BlackParrot: An agile open-source RISC-V multicore for accelerator SoCs. *IEEE Micro*, 40(4): 93–102, 2020.

[97] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced large-reach TLBs. In *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–269, 2012.

[98] Filip Pizlo. Locking in WebKit. https://webkit.org/blog/6161/locking-in-webkit/, 2016. Accessed: 2021-09-22.

[99] DPDK Project. Data Plane Development Kit. https://www.dpdk.org/, 2022. Accessed: 2022-08-07.

[100] Xiaogang Qiu and Michel Dubois. The synonym lookaside buffer: A solution to the synonym problem in virtual caches. *IEEE Transactions on Computers*, 57(12): 1585–1599, 2008.

[101] RISC-V International. riscv-tests: unit tests for RISC-V processors. https://github.com/riscv-software-src/riscv-tests, 2021. Accessed: 2021-11-08.

[102] Luigi Rizzo. netmap: a novel framework for fast packet I/O. In *21st USENIX Conference on Security Symposium*, pages 101–112, 2012.

[103] Alberto Ros, Mahdad Davari, and Stefanos Kaxiras. Hierarchical private/shared classification: the key to simple and efficient coherence for clustered cache hierarchies. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 186–197, 2015.

[104] Ali Saidi and Andreas Sandberg. gem5 virtual machine acceleration. http://www.m5sim.org/wiki/images/c/c3/2012_12_gem5_workshop_kvm.pdf, 2012.

[105] Yakun Sophia Shao, Sam Xi, Viji Srinivasan, Gu-Yeon Wei, and David Brooks. Toward cache-friendly hardware accelerators. In *HPCA Sensors and Cloud Architectures Workshop (SCAW)*, pages 1–6, 2015.

[106] Debendra Das Sharma. Compute express link®: An open industry-standard interconnect enabling heterogeneous data-centric computing. In *IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 5–12, 2022.

[107] Taku Shimosawa, Hiroya Matsuba, and Yutaka Ishikawa. Logical partitioning without architectural supports. In *32nd Annual IEEE International Computer Software and Applications Conference*, pages 355–364, 2008.

[108] SiFive. TileLink specification 1.8.1. https://sifive.cdn.prismic.io/sifive/7bef6f5c-ed3a-4712-866a-1a2e0c6b7b13_tilelink_spec_1.8.1.pdf, 2020. Accessed: 2021-09-24.

[109] Daniel J Sorin, Mark D Hill, and David A Wood. *A primer on memory consistency and cache coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[110] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In *Annual International Cryptology Conference*, pages 570–596, 2017.

[111] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. CAPI: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7–1, 2015.

[112] David Suggs, Mahesh Subramony, and Dan Bouvier. The AMD "Zen 2" processor. *IEEE Micro*, 40(2):45–52, 2020.

[113] Tuan Ta, Lin Cheng, and Christopher Batten. Simulating multi-core RISC-V systems in gem5. In *Second Workshop on Computer Architecture Research with RISC-V*, 2018.

[114] The Rust Team. Rust programming language. https://www.rust-lang.org/, 2020. Accessed: 2020-04-14.

[115] Linus Torvalds. test-tlb: Stupid memory latency and TLB tester. https://github.com/torvalds/test-tlb, 2018. Accessed: 2022-05-05.

[116] Tran Van Dung, Ittetsu Taniguchi, and Hiroyuki Tomiyama. Cache simulation for instruction set simulator QEMU. In *IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 441–446, 2014.

[117] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 205–218, 2010.

[118] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H Loh, and Abhishek Bhattacharjee. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 161–171, 2016.

[119] Wen-Hann Wang, J-L Baer, and Henry M Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *ACM/IEEE 16th Annual International Symposium on Computer Architecture (ISCA)*, 1989.

[120] Andrew Waterman and Krste Asanović. The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10, 2017.

[121] Kenneth C Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16 (2):28–41, 1996.

[122] Hongil Yoon and Gurindar S Sohi. Revisiting virtual L1 caches: A practical design using dynamic synonym remapping. In *IEEE 22nd International Symposium on High Performance Computer Architecture (HPCA)*, pages 212–224, 2016.

[123] Hongil Yoon, Jason Lowe-Power, and Gurindar S Sohi. Filtering translation bandwidth with virtual caching. In *23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 113–127, 2018.

[124] Florian Zaruba and Luca Benini. Ariane: An open-source 64-bit RISC-V application class processor and latest improvements. In *Technical talk at the RISC-V Workshop*, 2018.

[125] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. SonicBOOM: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*, volume 5, 2020.

[126] Yiren Zhao, Xitong Gao, Xuan Guo, Junyi Liu, Erwei Wang, Robert Mullins, Peter YK Cheung, George Constantinides, and Cheng-Zhong Xu. Automatic generation of multi-precision multi-arithmetic CNN accelerators for FPGAs. In *International Conference on Field-Programmable Technology (ICFPT)*, pages 45–53, 2019.

[127] Tianhao Zheng, Haishan Zhu, and Mattan Erez. SIPT: Speculatively indexed, physically tagged caches. In *IEEE 24th International Symposium on High Performance Computer Architecture (HPCA)*, pages 118–130, 2018.