# Architectural Implications of Automatic Parallelization With HELIX-RC

# Architectural Implications of Automatic Parallelization with HELIX-RC


A DISSERTATION PRESENTED
BY
KEVIN MATTHEW BROWNELL
TO
THE SCHOOL OF ENGINEERING AND APPLIED SCIENCES


IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
ENGINEERING SCIENCES


HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
SEPTEMBER 2015

Thesis advisor: Professor David M. Brooks, Gu-Yeon Wei Kevin Matthew Brownell

# Architectural Implications of Automatic Parallelization with HELIX-RC

## Abstract

As classic Dennard process scaling fades into the past, power density concerns have driven modern CPU designs to de-emphasize the pursuit of single-thread performance, focusing instead on increasing the number of cores in a chip. Computing throughput on a modern chip continues to improve, since multiple programs can run in parallel, but the performance of single programs improves only incrementally. Many compilers have been designed to automatically parallelize sequentially written programs by leveraging multiple cores for the same task, thereby enabling continued single-thread performance gains. One such compiler is HELIX, which can increase the performance of a mixture of SPECfp and SPECint benchmarks by 2× on a 6-core Nehalem CPU.

Previous approaches to automatically parallelize irregular programs have focused on removing apparent dependences through thread-level speculation, which limits the type of code that can be targeted. In contrast, this dissertation increases the amount of code that can be parallelized by addressing the specific communication demands of that code. The dissertation proposes a special-purpose extension of the cache hierarchy, called *ring cache*, to greatly reduce the perceived communication latency between cores running an automatically parallelized program. This co-design of ring cache and the HELIX compiler, called HELIX-RC, increases the speedup of 10 SPEC benchmarks running on 16 simulated in-order cores from an average of 2× to an average of over 8×. Speedups are slightly reduced to 7× on out-of- order cores, which extract instruction-level parallelism on their own. A fully synthesized Verilog implementation of ring cache is evaluated and is

shown to consume less than 25mW of power with an area of less than 0.275 square millimeters.

This dissertation includes a study comparing single program per core multiprogramming and HELIX-RC. Counterintuitively, some HELIX-RC parallelized benchmarks not only surpass simple multiprogramming in terms of single program performance, but can also beat multiprogramming in terms of total multicore throughput by reducing the effective per-core working set of a program.

With communication bottlenecks removed by ring cache, automatic parallelization with HELIX-RC restores a decade of lost single-thread performance improvements.

# Contents


0 Introduction 1

0.1 Performance Scaling Hits a Speed Bump . . . 2
0.2 Extracting Multicore Performance . . . 4
0.2.1 Single-Program Parallelism . . . 5
0.2.2 Multiple-Program Parallelism . . . 5
0.3 Core Utilization Remains Low . . . 6
0.4 Automatic Parallelization Can Improve Utilization . . . 6
0.5 Contribution of the Dissertation . . . 7
0.6 Organization of the Dissertation . . . 8

1 Prior Parallelization of Irregular Workloads Limited by Loop Size 9

1.1 Thread Extraction Techniques . . . 12
1.1.1 Cyclic Multithreading . . . 12
1.1.2 Pipelined Multithreading . . . 17
1.2 Speculation and Additional Hardware For Increasing Performance . . . 21
1.2.1 Software Speculation . . . 22
1.2.2 Hardware Speculation . . . 24
1.2.3 Custom Architectures . . . 28
1.3 Automatic Parallelization of Irregular Programs Must Handle Small Loops . . . 30
1.3.1 Hardware Requirements for Parallelizing Small Loops . . . 31

2 Existing Hardware Cannot Handle Requirements of Small Loops 33

2.1 Cache Coherence Protocols . . . 34
2.2 Scalar Operand Networks . . . 39
2.2.1 Tile Processor STN . . . 39
2.2.2 TRIPS OPN . . . 40
2.3 User-Controlled On-Chip Networks . . . 41
2.3.1 The Cell Processor Ring Network . . . 41

2.3.2 Tile Processor UDN . . . 42
2.4 Other Hardware for Accelerating Communication . . . 43
2.4.1 Multiscalar's Distributed Register File . . . 44
2.5 Conclusion . . . 46

3 Automatic Parallelization of Irregular Programs with HELIX-RC 48
3.1 Background and Opportunities . . . 50
3.1.1 Limits of Compiler-only Improvements . . . 50
3.1.2 Opportunity . . . 52
3.2 The HELIX-RC Solution . . . 57
3.2.1 Approach . . . 57
3.2.2 Decoupling Communication From Computation . . . 58
3.3 Compiler . . . 61
3.4 Architecture Enhancements . . . 63
3.4.1 Ring Cache Architecture . . . 64
3.4.2 Memory Hierarchy Integration . . . 67
3.5 Evaluation . . . 68
3.5.1 Experimental Setup . . . 68
3.5.2 Speedup Analysis . . . 72
3.5.3 Sensitivity to Architectural Parameters . . . 75
3.5.4 Analysis of Overhead . . . 76
3.6 Conclusion . . . 77

4 Ring Cache Detail and Implementation 78
4.1 Memory Hierarchy Integration . . . 79
4.1.1 Request and Reply Networks . . . 81
4.1.2 Reducing Remote Loads . . . 82
4.2 Signal Buffer Implementation . . . 83
4.2.1 Synchronization Epochs . . . 83
4.2.2 Signal Buffer Architecture . . . 90
4.2.3 Signal Buffer Optimization . . . 90
4.3 Ring Cache Synthesis Evaluation . . . 93
4.3.1 Signal Buffer Parameter Sweeps . . . 96
4.4 Conclusion . . . 101

5 Future Directions for HELIX-RC 102
5.1 HELIX-RC With Out-of-Order Cores . . . 103
5.1.1 Out-of-Order Execution . . . 104
5.1.2 Speedup Degradation in Out-of-Order Cores . . . 106
5.2 HELIX-RC vs. Multiprogram Parallelism . . . 112
5.2.1 Experimental Setup . . . 113
5.2.2 Evaluation . . . 115
5.3 Potential HELIX-RC Research Opportunities . . . 119
5.3.1 Compiler Engineering Improvements . . . 119
5.3.2 Compiler Sweeps . . . 119
5.3.3 Multiple-Loop Execution Model . . . 121

6 Conclusion 123

Appendix A Ring Cache Technical Report 124
A.1 Introduction . . . 124
A.2 Background . . . 125
A.2.1 HELIX Execution Model . . . 126
A.2.2 Parallel Code . . . 128
A.2.3 Decoupling Data Communication . . . 129
A.2.4 Decoupling Signal Forwarding . . . 133
A.3 Ring Cache Overview . . . 137
A.3.1 Core–Node Interaction . . . 138
A.3.2 Node to Node Connection . . . 140
A.3.3 Memory Hierarchy Integration . . . 142
A.4 Ring Cache Implementation . . . 143
A.5 Datapath Overview . . . 144
A.6 External Interfaces . . . 145
A.6.1 Core Interface . . . 145
A.6.2 L1 Cache Interface . . . 151
A.7 Network Interfaces . . . 154
A.7.1 Credit Based Flow Control . . . 156
A.7.2 Buffer Module . . . 159
A.8 Memory Flushing . . . 162
A.9 Storing Shared Data and Signals - The Forwarding Network . . . 165

A.9.1 Network Bundle . . . 168
A.9.2 Bundleizer Module . . . 168
A.9.3 Stopper Module . . . 172
A.10 Loading Shared Data - The Request/Reply Networks . . . 173
A.10.1 Request and Reply Networks . . . 176
A.10.2 Load Unit Module . . . 180
A.11 Ring Cache Memory . . . 192
A.11.1 Memory Module . . . 195
A.11.2 Array Module . . . 204
A.11.3 Bloom Filter Module . . . 207
A.12 Signal Buffer . . . 212
A.12.1 Synchronization Epochs . . . 212
A.12.2 Signal Buffer Module . . . 219
A.12.3 Signal Tracker Module . . . 226
A.12.4 Core Tracker Module . . . 227
A.12.5 Signal Buffer Optimizations . . . 232
A.12.6 Previous Implementations . . . 234
A.13 OS/Multiprogramming Considerations . . . 235
A.14 Synthesis Results . . . 236
A.14.1 Reference Design . . . 236
A.14.2 Signal Buffer Parameter Sweeps . . . 240

Appendix B Ring Cache Verilog Code 245
B.1 defines.v . . . 246
B.2 ring_cache.v . . . 248
B.3 buffer.v . . . 258
B.4 bundleizer.v . . . 262
B.5 stopper.v . . . 266
B.6 load_unit.v . . . 268
B.7 memory.v . . . 275
B.8 priority_encoder.v . . . 285
B.9 array.v . . . 286
B.10 bloom_filter.v . . . 290
B.11 hash.v . . . 292

B.12 signal_buffer.v . . . . . 293
B.13 signal_buffer_signal_tracker.v . . . . . 296
B.14 signal_buffer_core_tracker.v . . . . . 299

References 309

# Listing of figures


1 Historical clock frequency scaling trend. . . . . 2
2 Historical single-threaded performance scaling. . . . . 3
3 Number of cores on a single die. . . . . 4

1.1 A candidate loop for DOACROSS parallelization . . . . . 13
1.2 Decomposition of loop iteration by DOACROSS . . . . . 13
1.3 A loop schedule for DOACROSS parallelization . . . . . 15
1.4 A loop schedule for DOACROSS parallelization with high communication latency . 16
1.5 A loop schedule for DOACROSS parallelization with a small parallel region . . . . . 17
1.6 Decomposition of loop iteration by HELIX . . . . . 18
1.7 A loop schedule for HELIX parallelization with a small parallel region . . . . . 18
1.8 Decomposition of loop iteration by DSWP . . . . . 19
1.9 A loop schedule for DSWP parallelization with unbalanced stages . . . . . 20
1.10 A loop schedule for DSWP parallelization with balanced stages . . . . . 21

2.1 Decomposition of loop iteration by HELIX with synchronization instructions . . . 36
2.2 HELIX communication penalty with reactive data transfer . . . . . 37
2.3 HELIX communication penalty with hypothetical proactive data transfer . . . . . 38
2.4 Multiscalar Distributed Register File . . . . . 44

3.1 Augmenting the HELIX compiler does not improve irregular program performance 51
3.2 Accuracy of dependence analysis for small hot loops in irregular benchmarks . . . . 54
3.3 Short loop iterations in SPECint 2000 . . . . . 54
3.4 Predictability of variables reduces register communication in small hot loops . . . . 55
3.5 Distribution of required communication distance between 16 cores . . . . . 55
3.6 Most shared data is consumed by multiple cores . . . . . 56
3.7 Example of decoupled data and signal communication. . . . . 60
3.8 Ring cache architecture overview . . . . . 64
3.9 HELIX-RC triples the speedup obtained by HCCv2 . . . . . 70

3.10 Breakdown of benefits of decoupling communication from computation . . . . . 71
3.11 Code generated assuming the existence of ring cache slows down on normal hardware 73
3.12 Speedup sensitivity to core count and ring cache parameters . . . . . 75
3.13 Breakdown of overheads that prevent HELIX-RC from achieving ideal speedup . . 75

4.1 Ring cache must be carefully integrated the normal cache hierarchy . . . . . 80
4.2 An empty sequential segment is protected only be a *light wait* . . . . . 84
4.3 Cores constrained to a single epoch have reduced performance . . . . . 87
4.4 Cores that can decouple by an additional epoch have higher performance . . . . . 89
4.5 Signal Buffer architecture . . . . . 91
4.6 Synthesized ring cache power and area . . . . . 96
4.7 Total ring node area as total signal ID capacity is swept from 8 to 512. . . . . 97
4.8 Increasing signal bandwidth increases signal buffer and network buffer sizes. . . . . 98
4.9 Increased decoupling increases ring cache area . . . . . 99
4.10 Speedups plateau at two epochs of decoupling . . . . . 99
4.11 Ring cache area reduces when there are fewer cores in the system . . . . . 100

5.1 Singled-threaded SPECint 2000 performance on different core types . . . . . 103
5.2 HELIX-RC SPEC CPU2000 speedups on different core types . . . . . 104
5.3 Overall HELIX-RC performance always increases for higher performance cores . . . 105
5.4 Performance bottlenecks on a single sequential segment . . . . . 107
5.5 Program latency vs. multicore throughput for 183.equake . . . . . 115
5.6 Program latency vs. multicore throughput for 179.art . . . . . 116
5.7 Program latency vs. multicore throughput for 188.ammp . . . . . 116
5.8 Program latency vs. multicore throughput for 197.parser . . . . . 117
5.9 Program latency vs. multicore throughput for 164.gzip . . . . . 118
5.10 HELIX's memory dependence analysis encounters diminishing returns . . . . . 120
5.11 Splitting sequential segments improves HELIX-RC speedups up to a certain point . 120

A.1 HELIX execution model . . . . . 127
A.2 A HELIX paralellized loop . . . . . 128
A.3 Reactive communication produces worse performance than proactive communication 130
A.4 Shared data is often accessed by an unpredictable number of cores . . . . . 131
A.5 Decoupling data and synchronization communication is vital for speedups . . . . . 132
A.6 Sequential forwarding chains limit HELIX-style parallelization . . . . . 134

A.7 Breaking sequential forwarding chains improves parallel performance . . . . . 135
A.8 Ring cache architecture overview . . . . . 137
A.9 Schematic of top level ring cache module. . . . . 146
A.10 A ring node has direct connections to its local core and its local L1 cache. . . . . 147
A.11 Load timing diagram for ring cache hit . . . . . 150
A.12 Load from ring node to L1 cache . . . . . 151
A.13 A ring node is connected with its neighbor ring node by three different networks . . 155
A.14 Schematic of the buffer module . . . . . 160
A.15 Control FSM for buffer module . . . . . 161
A.16 Ring cache flush timing diagram . . . . . 163
A.17 Forwarding network bundle . . . . . 165
A.18 Schematic of bundleizer module . . . . . 166
A.19 Schematic of stopper module . . . . . 167
A.20 Incorrect ring cache memory hierarchy integration . . . . . 174
A.21 Request/reply network bundles . . . . . 176
A.22 Schematic of load unit module . . . . . 181
A.23 Load unit FSM for local loads . . . . . 187
A.24 Load unit FSM for remote loads . . . . . 188
A.25 Cache bits and owner bits for a ring cache memory address . . . . . 192
A.26 Schematic of memory module . . . . . 196
A.27 Memory module FSM for loads . . . . . 202
A.28 Memory module FSM for stores and flushes . . . . . 203
A.29 Schematic of array module . . . . . 204
A.30 Schematic of bloom filter module . . . . . 208
A.31 Empty sequential segments are protected by modified light waits . . . . . 213
A.32 One bit of signal buffering allows cores to decouple by only one epoch . . . . . 214
A.33 Two bits of signal buffering allows cores to decouple by two epochs . . . . . 215
A.34 Schematic of signal buffer . . . . . 220
A.35 Signal entry bits . . . . . 220
A.36 Schematic of signal tracker module . . . . . 221
A.37 Schematic of core tracker module . . . . . 222
A.38 Core tracker FSM . . . . . 223
A.39 Core tracker module initialization . . . . . 223
A.40 Power and area for a single ring node . . . . . 239

A.41 Total ring node area as total signal ID capacity is swept from 8 to 512. . . . . 240
A.42 Sensitivity of signal bandwidth on speedup . . . . . 241
A.43 Increasing signal bandwidth increases the signal buffer and network buffer sizes. . . 242
A.44 Decoupling synchronization from one to two epochs increases area significantly . . . 242
A.45 Decoupling synchronization up to two epochs increases speedups . . . . . 243
A.46 HELIX-RC scales relatively well on a small number of cores. . . . . 244
A.47 Signal buffer area is linear with number of supported cores . . . . . 244

# Listing of tables


3.1 Characteristics of parallelized benchmarks. . . . . 69

4.1 Ring cache parameters for the reference design . . . . . 93
4.2 Synthesis results for a single reference ring node. . . . . 95

5.1 Working set sizes for SPECint 2000. . . . . 114

A.1 Ring Cache parameters for the reference design. . . . . 237
A.2 Synthesis results for a single reference ring node. . . . . 238

# Previous Work

## Portions of this dissertation have appeared in:

Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. "HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs." In Proceeding of the 41st annual international symposium on Computer architecuture, pp. 217-228. IEEE Press, 2014.

and

Kevin Brownell. "Ring Cache Technical Report."

## Portions of this dissertation are in submission to:

ACM Transactions on Architecture and Code Optimization (TACO)

# 0

# Introduction

In 1965, Gordon Moore observed that the number of transistors per unit area on an integrated circuit was increasing by a factor of two year after year [42]. Although the timeframe of his original forecast was not entirely correct, Moore's Law heralded the general trend of regular doublings of transistor density as the semiconductor industry strived to integrate more and more transistors. After nearly 50 years of process technology improvements, the number of transistors in a chip has increased from hundreds in the 1960s to well over a billion today. Along with the explosion in the transistor count came a seemingly relentless increase in computing performance. A portion of this increase was due to CPU architecture improvements, but a larger portion was the result of faster and faster transistors [5].

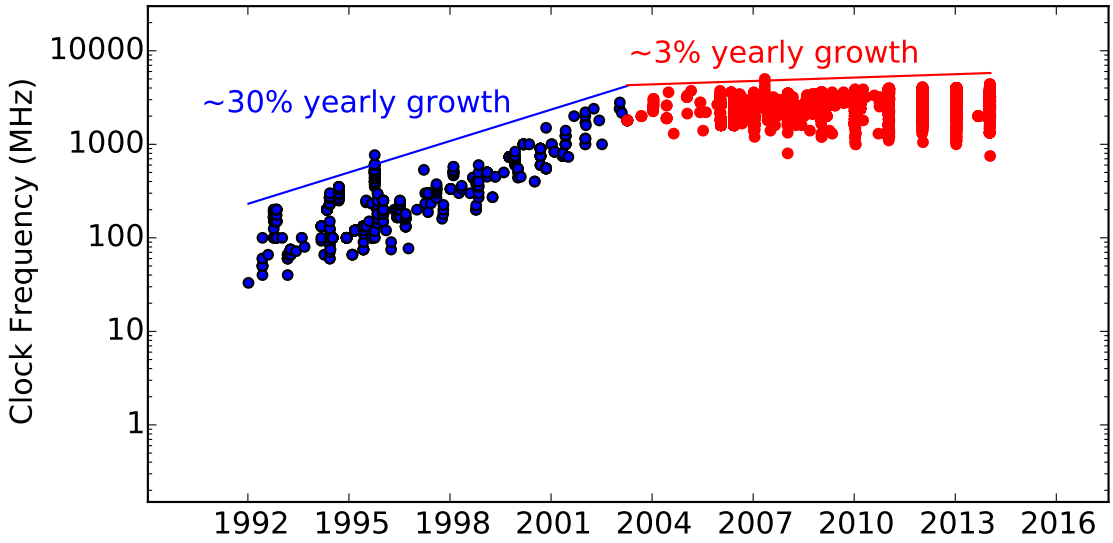



**Figure 1:** Due to a breakdown of Dennard scaling, the growth rate of nominal CPU clock frequencies dramatically decreased about a decade ago. Rising power density made further increases infeasible. Historical data from the Stanford CPUDB project [17].

For decades, smaller transistor sizes provided what seemed to be a free lunch. As feature sizes decreased by a predictable factor, so too did capacitance. By decreasing the supply and threshold voltages by the same factor, the speed of transistors could also be increased and their power consumption decreased. Through this process, known as Dennard scaling [18], the power density of a chip remained constant as clock frequencies steadily increased. This resulted in reliable single-threaded performance increases—every new process technology meant that the previous year's programs now ran faster.

## 0.1 Performance Scaling Hits a Speed Bump

Unfortunately, in the early 2000s, Dennard scaling began to break down. Due to increasing amounts of leakage current, the previously steadily decreasing threshold voltage began to plateau [36]. Consequently, clock speeds could not continue to increase, or so too would the power density of a chip. Given limitations in the ability to cool a chip beyond a certain power ceiling, the industry had no

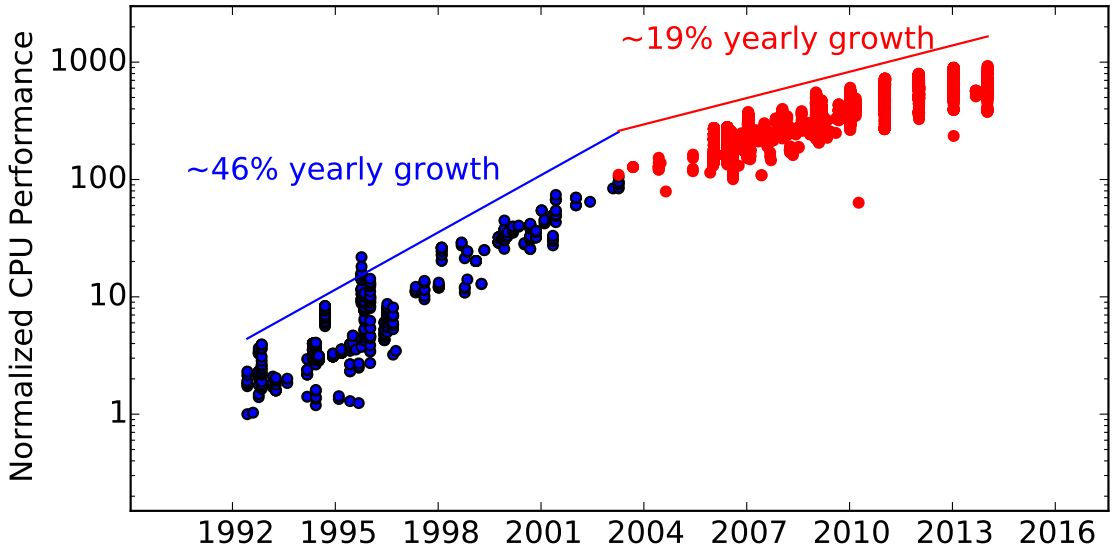



**Figure 2:** The "power" wall resulted in sharply decreased historical single-threaded performance gains. CPU performance has been normalized across multiple generations of the SPECint benchmark suite. Only CPUs from the database with SPECint numbers are plotted. Historical data from the Stanford CPUDB project [17].

choice but to significantly reduce the aggressiveness of clock frequency increases. Figure 1 shows the dramatic slowdown in clock frequency gains that resulted from the breakdown of Dennard scaling. Nominal CPU frequencies plateaued around 2004, in the 3–4 GHz range.

Hand in hand with stalls in clock frequency gains, single-thread performance gains stalled as well. Figure 2 shows normalized single-threaded performance around this time period for a large variety of CPUs. The performance data, taken from the Stanford CPUDB project [17], has been normalized across multiple generations of the SPECint benchmark suite. Prior to 2004, when clock frequencies were still increasing, overall single-threaded CPU performance increased by nearly 46% per year. After Dennard scaling broke down, performance still increased, but at the much slower rate of 19% per year. Had the original 46% trend continued past 2004, CPU performance would be 5–10× higher today. With this "power wall" blocking single-threaded performance gains, the industry decided to instead use their still growing transistor budget to integrate multiple identical general-purpose cores on a single die. Figure 3 shows the dramatic increase in the number of cores

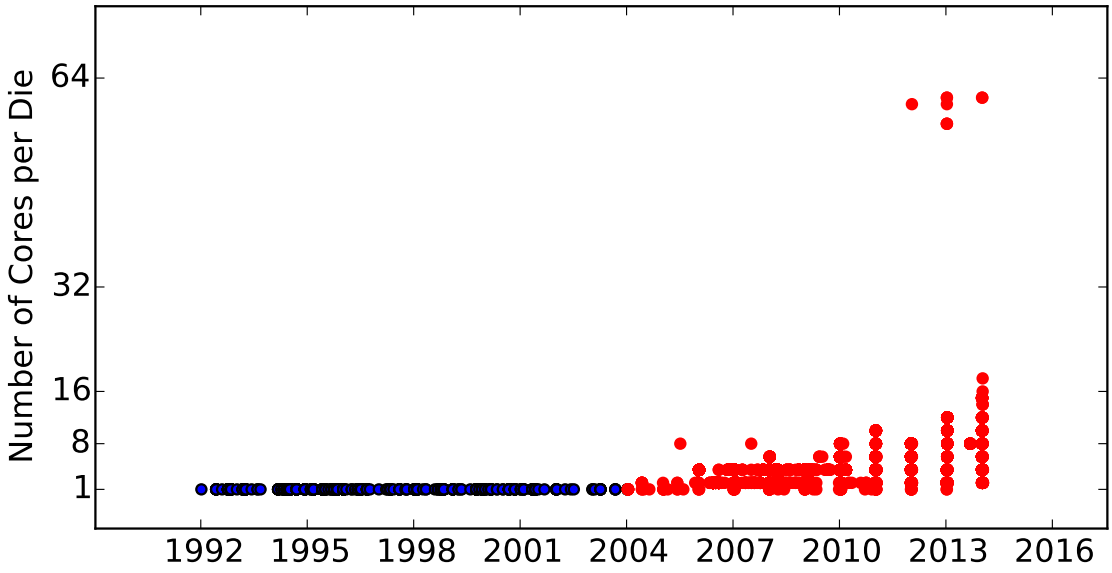



**Figure 3:** Facing the power wall, industry transitioned to placing multiple cores on a single die. Historical data from the Stanford CPUDB project [17], representing only general purpose CPUs.

starting in 2004. Since clock frequency ultimately has a cubic relationship with power, multiple cores clocked at lower frequencies may still fit within a fixed power budget while at the same time providing higher theoretical computing performance. This higher performance can only be realized, however, if the multiple cores can all be utilized simultaneously, at least for some fraction of the time.

## 0.2 Extracting Multicore Performance

Broadly, there are two primary ways to extract performance from a general-purpose multicore processor. First, a single program can be decomposed into different execution threads to exploit thread-level parallelism (TLP) to gain performance on multiple cores. Alternatively, multiple programs can independently use different cores at the same time. Although these techniques are usually easily applicable to simple and regular programs, they are often lacking for irregular workloads that contain complex data and control flows.

### 0.2.1 Single-Program Parallelism

Although parallel computing had already long existed, the introduction of multiple cores in a single chip opened the door for finer-grained parallel computing, which previously had been limited to workloads that could tolerate the long communication latencies between different chips and machines. Decomposing a program into multiple threads, each of which can run on a different core, can significantly improve the performance of a single program running on a multicore chip. An increasing variety of modern tools and programming models have been introduced to facilitate multithreaded programming. Depending on workload complexity and available programmer time as well as programmer ability, some programs are easier to split into threads than others. In general, multithreaded programs are much more difficult to create, maintain, and debug than single-threaded programs. Additionally, it is often difficult for programmers to create balanced amounts of work for the threads, so that the realized performance increase from multithreading is often far less than the theoretical performance increase. As a result, programmers often feel that it is not worth the effort needed to make a program parallel, and they tend instead to rely entirely on the slower single-thread performance scaling to gain performance.

### 0.2.2 Multiple-Program Parallelism

An easier way to extract multicore performance is through the use of multiple-program parallelism: instead of trying to parallelize a single program, multiple programs can be run on different cores at the same time. Even though single-threaded performance does not increase, the total throughput does, so multicore computing resources are not wasted. Multiple-program parallelism also has the benefit of scaling relatively well as long as the multiple programs do not interact destructively. As more cores are added to a chip, the total throughput may increase by a predictable amount. Unfortunately, the large amount of shared resources on a multicore processor (shared caches, DRAM

bandwidth, on-chip network bandwidth, etc.) can result in less than ideal throughput scaling.

## 0.3 Core Utilization Remains Low

Both single-program and multiple-program parallelism have been insufficient for keeping multicore utilization high. For datacenter-scale computing, core utilization is usually well below 50% on average [3]. One reason for the low utilization is the desire to isolate latency-sensitive applications, so that processors are intentionally underprovisioned to ensure that multiple programs don't overly contend for shared resources [39]. Another reason is the desire to ensure that spare computing capacity is available if demand increases. Either way, the result is that cores sit idle.

In the mobile realm (e.g., phones and tablets), core utilization is dramatically lower than one would expect, considering the ever-increasing number of cores generation after generation. Studies have shown that although popular applications tend to have some TLP, most mobile applications use less than two cores on average [20]. Additionally, typical mobile device interactions generally encourage use of only a single application at a time, so multiple-program parallelism is also limited. The theoretical performance from having up to 8 cores on a single mobile device is thus largely wasted.

## 0.4 Automatic Parallelization Can Improve Utilization

Given the difficulty of manually extracting TLP from sequential code, automatic parallelization offers a promising route for increasing multicore utilization. Not only can automatic thread extraction make use of an increasing number of cores; it can also increase single-program performance. Historically, a variety of techniques sought to parallelize programs across multiple chips and/or multiple machines by automatically extracting threads [16, 28]. Each of these extracted threads would run on a different processor/machine, and they would communicate when necessary for synchronization or sharing data. While these techniques realized some success, they were as a rule only applicable

to workloads that had minimal communication or synchronization requirements, generally those with very regular control and data flow. Due to the large latency between chips and machines, if a program required frequent or irregular communication, the time spent communicating would dominate the total execution time.

As the multicore era took hold, there was renewed interest in leveraging automatic parallelization to regain lost single-thread performance. With multiple cores close together on a die, communication costs decrease and inter-core bandwidth increases, making previously unscalable techniques more realistic for a larger variety of workloads. A growing number of compiler techniques to extract threads have proved to be feasible for previously unparallelizable irregular programs, most of these techniques variations of either cyclic-multithreading or pipelined-multithreading parallelism [12, 46, 48, 60]. Efforts have also been made to extract parallelism by combining compiler techniques with custom hardware [25, 38, 54, 53, 56], with some success. Despite this revitalized interest, however, there is still much room for improvement—specifically, there is a need for a technique that 1) is broadly applicable to a large number of irregular programs, 2) produces high speedups on those programs, and 3) doesn't require large changes to existing general-purpose multicore architectures.

## 0.5 Contribution of the Dissertation

In this dissertation, I first examine a recent compiler technique for automatic parallelization called HELIX [12], detailing its intrinsic performance limitations and bottlenecks. In order to boost the performance of HELIX, I propose a co-design comprising an improved version of HELIX and a light-weight hardware extension. This co-design, called HELIX-RC, boosts the speedup of sequentially written irregular code—that is, code that contains complex data and control flows—from 2× to 6.85×, which buys back a large portion of the single-threaded performance gains lost over the last 10 years. The speedup improvements stem from the ability of the co-designed compiler and hard-

ware to extract parallelism from loops with much higher communication requirements than prior initiatives have been able to address. Moreover, by efficiently utilizing non-core resources, HELIX-RC can achieve higher multicore throughput even in cases where multiple-program parallelism is already abundant. The additional hardware component, ring cache, is easily integrated into existing commodity multicore architectures at minimal power and area costs. The architectural implications and the implementation of ring cache are evaluated in detail.

## 0.6 Organization of the Dissertation

The rest of this dissertation is organized as follows. First, Chapter 1 details relevant historical and modern automatic parallelization techniques, with an emphasis on their limitations with respect to parallelizing irregular workloads. The characteristics of the hardware support needed to boost the performance of these workloads are described. Next, Chapter 2 presents existing hardware mechanisms for inter-core communication and explains why existing hardware fails to address the communication needs of irregular programs. Chapter 3 details and evaluates the proposed compiler-architecture co-design, which is a combination of the HELIX compiler and some novel hardware, the ring cache. Chapter 4 presents selected implementation details for ring cache, in addition to synthesis results from a cycle-accurate Verilog model of the hardware. The full ring cache implementation report appears in Appendix A. Finally, Chapter 5 examines some architectural tradeoffs regarding HELIX-RC, including its potential effect on different core architectures and its use for tradeoffs between program execution time and overall multicore processor throughput, along with possible future compiler extensions to further increase performance.

# 1

# Prior Parallelization of Irregular Workloads Limited by Loop Size

While some computing problems often translate to either inherently parallel or easy-to-parallelize numerical programs, sequentially designed, irregular programs with complicated control (e.g., execution paths) and data flows (e.g., aliasing) are much more common but difficult to analyze precisely. For years, many attempts have been made to accelerate single-thread performance beyond what has been provided by traditional process scaling and architectural improvements. Although the conventional wisdom is that irregular programs cannot make good use of multiple cores, research in the past decade has made steady progress towards extracting TLP from complex, sequen-

tially designed programs such as the integer benchmarks from the SPEC CPU suites.

Some of this past research has focused primarily on compiler techniques to automatically extract parallel threads from sequentially written code. Other work combines compiler techniques with special-purpose hardware in an attempt to overcome some of the limitations of the compiler-only strategies. In general, these strategies are most successful on so-called *regular* (or *numerical*) workloads—those with predictable control flow and data access. For *irregular* workloads, these techniques tend not to be so successful.

The two primary approaches for automatic thread extraction are cyclic multithreading and pipelined multithreading. Both operate by transforming sequentially written loops into multiple threads that run on different cores or, historically, on different machines. Inter-thread communication is used to satisfy any required synchronization or data dependence forwarding. Cyclic multithreading assigns different loop iterations to different threads. Any loop-carried dependence (i.e., a dependence between different loop iterations) is communicated between threads from older iterations to younger iterations, forming a cycle between the threads. In contrast, pipelined multithreading forms a pipeline between threads, rather than a cycle. A loop iteration is split into multiple stages (e.g., the first half of every iteration belongs to one stage and the second half of every iteration belongs to a stage), with each stage assigned to a different thread. Thus, unlike cyclic multithreading, every thread runs a portion of every iteration in pipelined multithreading. Although many variations of these techniques exist, the core transformation of loops into thread cycles or pipelines remains roughly the same.

For both categories of thread extraction, loops with larger iterations generally contain larger amounts of code that can run in parallel, with less required communication. Smaller loops tend to be more tightly coupled—and with such short loop bodies, the cost of performing any kind of communication can quickly dwarf any benefit of parallelization. Since small loops require at least some amount of communication, both cyclic multithreading and pipelined multithreading tend to

perform poorly for them, even though the loops may contain large amounts of potential parallelism. For this reason, most state-of-the-art parallelization techniques target relatively large loops.

Unfortunately, complex control and data flows in irregular programs—both exacerbated by ambiguous pointers and ambiguous indirect calls—make accurate data dependence analysis difficult. In addition to actual dependences that require communication between threads, a compiler must conservatively handle apparent dependences that are never realized at runtime. Additionally, larger loops are harder to analyze, due to the increased lexical scope and amount of variables/memory being considered. If all of the apparent dependences need to be synchronized, performance will suffer greatly.

A common way to handle a large number of apparent dependences is through speculation [35, 38, 56], which avoids the need for accurate data dependence analysis by speculating that some apparent dependences are not realized. However, thread-level speculation (TLS) suffers from the overhead needed to support misspeculation and therefore is primarily limited to targeting relatively large loops in order to amortize penalties.

A potential alternative strategy to existing parallelization solutions is to target small loops instead, as these are much easier to analyze via state-of-the-art control and data flow analysis, which significantly improve accuracy. Furthermore, this ease of analysis enables transformations that simply recompute shared variables in order to remove a large fraction of actual dependences. This strategy increases TLP and reduces core-to-core communication. Such optimizations do not readily translate to TLS because the complexity of TLS-targeted code typically spans multiple procedures in larger loops.

In the remainder of this chapter, I first discuss some of the primary compiler techniques for automatic thread extraction and detail their strengths and drawbacks, especially with regard to accelerating irregular programs. Next, I discuss refinements of these techniques that attempt to overcome some of the limitations of compiler-only solutions, such as compiler–architecture co-designs and

those that use TLS. Then I explore an opportunity to increase performance even further by targeting small loops, an untapped source of parallelism that has so far been left on the table. This opportunity is only realizable, however, if the significant communication requirements of small-loop parallelization are fulfilled.

## 1.1 Thread Extraction Techniques

There are two primary models for extracting parallel threads from sequentially written loops. The two approaches, cyclic multithreading and pipelined multithreading, underpin most automatic parallelization techniques. In the following subsections, I describe the general transformation of sequential code to parallel threads, as well as the primary performance bottlenecks and potential pitfalls of each technique.

### 1.1.1 Cyclic Multithreading

Cyclic multithreading was one of the first parallel processing paradigms to be introduced, in 1966 [4]. In general, cyclic multithreading (CMT) operates by assigning loop iterations to different threads, which are then executed on different cores or processors. Once a core $i$ completes iteration $i$, it next executes iteration $i + n$, where $n$ is the number of cores in the system (e.g., on a 4-core system, core 0 would execute iteration 0, then iteration 4, and so on). For simple loops that have no loop-carried dependences (a degenerate case of CMT, often called DOALL), the threads can more or less run independently. Unfortunately, other than relatively trivial or basic number-crunching scientific applications, the vast majority of programs contain loops with control and data dependences. For these nontrivial loops, the DOACROSS [16] strategy, which partitions iterations into a sequential portion and a parallel portion, was developed. The sequential portion contains any loop-carried dependences and must be executed in loop iteration order, in effect forming a cycle between threads as older iterations feed data to younger iterations. The parallel portion can be executed completely

```
Node* node = root;
int mySum = node->data;

for(int i = 0; i < 8; i++) {
     node = node->next;
     mySum = mySum + node->data;
     work(mySum, node->data);
}
```

**Figure 1.1:** A candidate loop for DOACROSS parallelization.

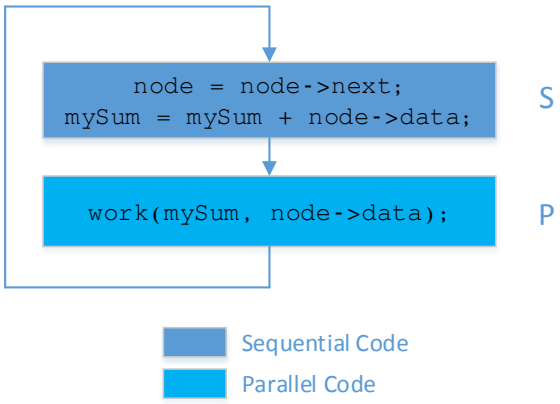



**Figure 1.2:** A loop iteration is decomposed into sequential and parallel portions for DOACROSS parallelization.

independently. More recently, a generalization of DOACROSS called HELIX [12] has further split the sequential portion of each iteration into multiple smaller sequential segments. This potentially enhances performance by enabling parallelism between different sequential segments.

## DOACROSS

To illustrate the DOACROSS transformation, consider the code example in Figure 1.1. This loop contains two loop-carried dependences. First, the *node* pointer for the linked list is updated by every iteration as the linked list is traversed. Second, the *mySum* variable contains the running sum of the data located at each node. Let us assume that the subsequent *work* function is completely independent and does not access any memory or registers shared between iterations. Figure 1.2 shows a

transformed version of an iteration of this loop. DOACROSS places the two loop-carried dependences into a sequential region, which must be executed in loop iteration order (enforced by synchronization instructions; not shown), and the independent *work* function (which relies only on values produced by the current iteration of the loop) into a parallel region. Execution of the parallel region can overlap with the sequential region of any younger iteration and the parallel region of any other iteration. An execution timeline for eight iterations of this loop on four cores is shown in Figure 1.3. The values of loop-carried dependences flow undirectionally from older iterations to younger iterations—from core 0 to core 1, 2, and 3, and then back to core 0, forming a cycle. Note that because the parallel region has a long execution time relative to the sequential portion, there is a significant performance gain from the large amount of code that can execute in parallel.

There are two primary bottlenecks that can severely limit the performance of DOACROSS. Consider Figure 1.4, where the time it takes for data to transfer between cores is tripled, leading to cores stalling as they wait for loop-carried data to transfer. Execution time in this scenario is much longer due to the tight coupling between cores intrinsic to CMT, making *communication latency* a significant factor for DOACROSS. The second primary bottleneck is the size of the sequential portion of the iteration relative to the parallel portion. If there is a large number of loop-carried dependences compared to the size of the independent code, there will be limited opportunities for overlap between threads, which in turn reduces performance. Figure 1.5 depicts an execution timeline with a much shorter *work* function, resulting in significant stalls. This highlights the importance of having a larger parallel-to-sequential code ratio.

Unfortunately, irregular workloads not only have a large number of actual loop-carried dependences (relative to regular workloads), but are also susceptible to a large number of apparent dependences (i.e., dependences that do not manifest at runtime), which bloat the size of the sequential portion of the loop iteration. In order to achieve good performance, DOACROSS must be able to keep the sequential portion small and must have very fast inter-thread communication.

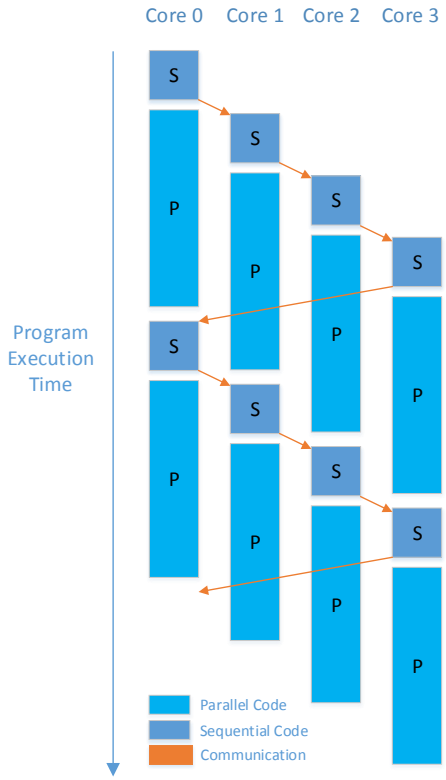



**Figure 1.3:** Four cores execute eight iterations of a DOACROSS loop, with data flowing from older iterations to younger iterations. Since the communication delay is small and the parallel region is large, all four cores have high utilization.

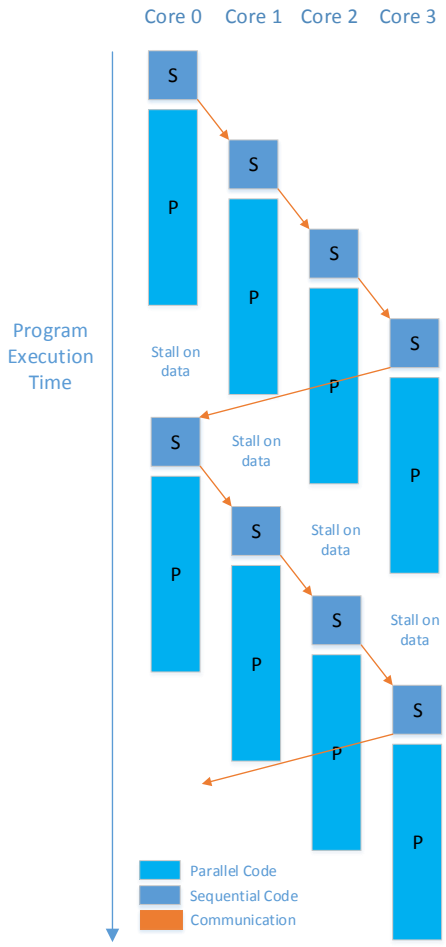



**Figure 1.4:** With slightly higher communication latency, communication stalls hurt DOACROSS performance.

## HELIX

HELIX [12], an evolution of DOACROSS, addresses the sequential portion bottleneck by splitting the sequential portion into multiple sequential segments (it may also create multiple parallel segments). Although each sequential segment still needs to run in loop iteration order, different segments can execute simultaneously, to exploit parallelism among them. Figure 1.6 shows how HELIX decomposes the original loop body shown in Figure 1.1. Note that there are now two different independent sequential segments, so and s1. Segment zero relies only on the previous iteration's

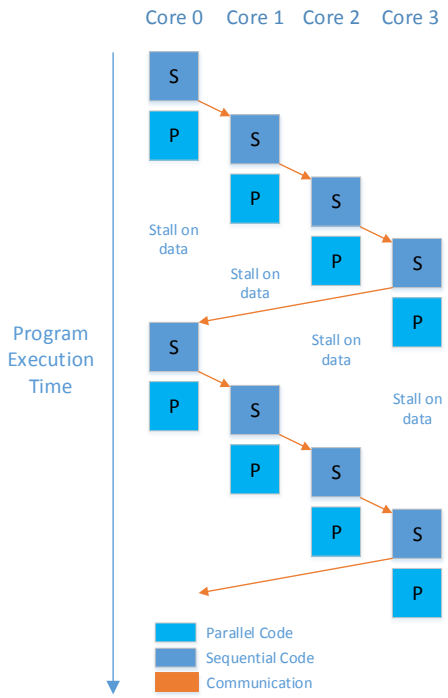



**Figure 1.5:** If the parallel region of a DOACROSS loop is short, there can potentially be severe performance degradation.

value for *node*, and segment one relies only on the previous iteration's value of *mySum* (it uses the current iteration's value of *node*). Even when the parallel region is small, as was the case in Figure 1.5, HELIX can improve performance by allowing so and s1 to overlap their execution, as shown in Figure 1.7. However, as with DOACROSS, HELIX□s speedups are limited by communication latency. Despite this remaining bottleneck, HELIX was able to achieve a speedup of 2.25× for a mixture of SPEC CPU 2000 integer and floating point benchmarks.

### 1.1.2 Pipelined Multithreading

In contrast to CMT, pipelined multithreading (PMT) splits loop iterations into different stages, each of which is then assigned to a single thread. The data dependences that need communicating in this case are not loop-carried, as with CMT—instead, they are intra-iteration dependences.

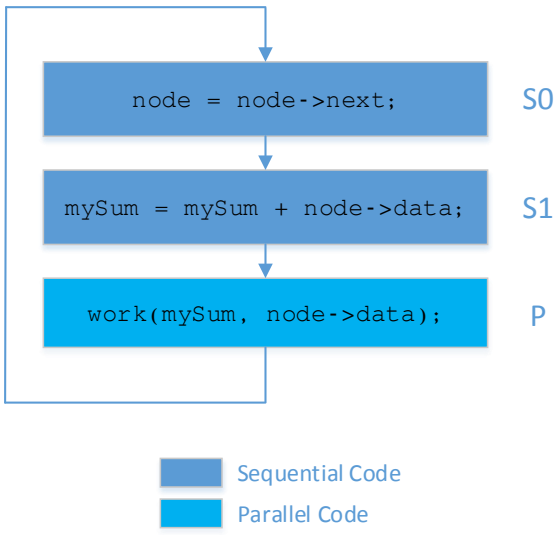



**Figure 1.6:** A loop iteration is decomposed into sequential and parallel portions for HELIX parallelization. DOACROSS would have created only one sequential portion.

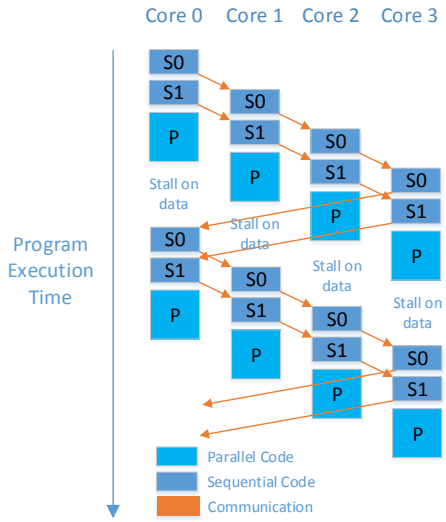



**Figure 1.7:** Even with a short parallel region, HELIX can reduce stalls by executing different sequential segments in parallel.

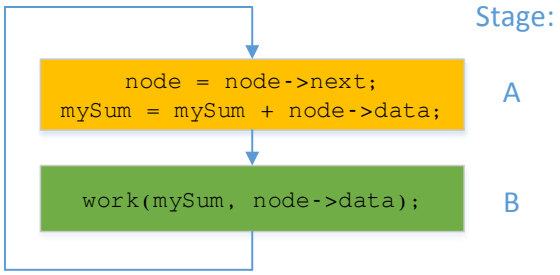



**Figure 1.8:** A loop iteration is decomposed into two stages for DSWP parallelization.

These stages create a pipeline where data flows from the first stage of the pipeline to the last. Unlike threads in CMT, each thread in PMT executes a portion of every iteration, so no cycle is formed, and the different threads are not so tightly coupled. The most preeminent example of this technique is known as decoupled software pipelining, or DSWP [46].

### DSWP

For the code example in Figure 1.1, DSWP may create two different pipeline stages, as shown in Figure 1.8. The first stage, A, encompasses the updates to the *node* and *mySum* variables. The second stage, B, includes just the *work* function. Figure 1.9 depicts an execution timeline with this organization for three iterations on a 2-core system. Since there are only two stages of the pipeline, only two cores can be used. Core 0 repeatedly executes stage A and then communicates the values of *mySum* and *node* to core 1, which repeatedly executes the *work* function on the incoming data.

Since PMT creates a pipeline, it is less sensitive than CMT to communication latency, which only affects the pipeline fill time. However, it has two other primary bottlenecks. First, as can be seen in the figure, if different stages of the pipeline are not well balanced, performance is limited by the longest stage and core utilization is very low. Figure 1.10 shows the much higher core utilization when the pipeline stages are balanced. The second major bottleneck is bandwidth: if many small pipeline stages are created, there might not be adequate inter-thread bandwidth (or buffering)

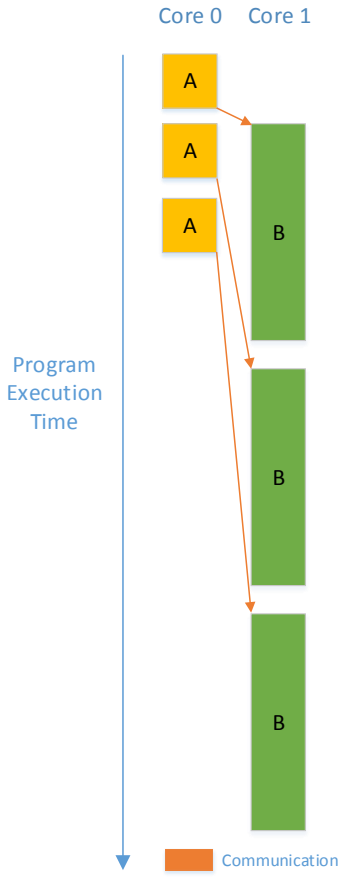



**Figure 1.9:** Two cores executing three iterations of a loop parallelized by DSWP suffer low utilization due to pipeline stage imbalance.

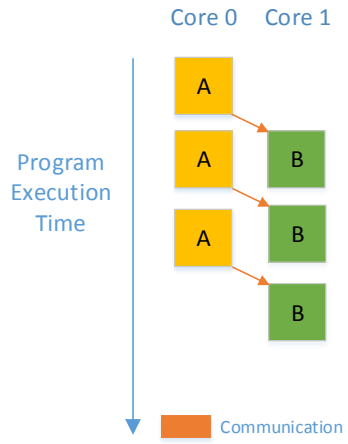



**Figure 1.10:** With balanced stages, DSWP offers a parallelization scheme that is robust to communication latency, but potentially sensitive to inter-core bandwidth.

to keep every stage of the pipeline busy. Irregular workloads with complex input-dependent control and data flows exacerbate this problem, as a compiler will then have more trouble predicting pipeline balance and bandwidth requirements at compile time. Depending on the complexity of the code and the amount of potential memory aliasing, there may be a very small number of possible options for pipeline stages. DSWP favors larger loops, since smaller loops may be more difficult to split into many balanced stages. In its original conception, DSWP achieved approximately a 1.1× program speedup on a variety of benchmarks, including some from SPEC CPU 2000 [46].

## 1.2 Speculation and Additional Hardware For Increasing Performance

We have seen that the fundamental CMT and PMT automatic parallelization approaches contain some intrinsic bottlenecks that limit their success in parallelizing irregular programs. For CMT, communication latency is the primary problem, whereas for PMT, communication bandwidth and pipeline balance are the primary problems. These limitations are exacerbated by a potentially large number of apparent dependences that a compiler must conservatively satisfy but that don't actually manifest at runtime. Additionally, since irregular programs tend to have unpredictable control and

data flows from iteration to iteration, there are a large number of dependences that only occasionally manifest but that nevertheless must be handled. Although TLS helps in cases where dependences do not manifest, misspeculation costs harm performance when dependences do manifest. Only loops with a high ratio of apparent to actual dependences can reliably achieve high speedups. Sometimes the compiler can prove that certain dependences will always manifest at runtime, and hardware can be used to accelerate the communication of these. However, especially for irregular programs, these are the minority of dependences, since memory references are often ambiguous.

Over the years, a large body of work has been devoted to improving irregular program performance beyond CMT and PMT: TLS techniques help reduce the performance impact of apparent dependences [25, 31, 35, 38, 41, 56, 61, 66, 67], and hardware support for decreasing communication costs helps reduce the performance impact of actual dependences [50, 53, 54]. Other work combines CMT and PMT to improve speedups [27, 49]. More radical co-designs of the compiler and the computer architecture holistically extract parallelism differently than vanilla CMT/PMT while reducing the impact of both apparent and actual dependences [51, 53, 54].

The rest of this section describes some recent improvements for automatically parallellizing programs. I will first discuss notable software-only speculation approaches, and then influential techniques that rely on hardware TLS. Finally, I detail an example of a significant compiler–architecture co-design for extracting parallelism. Although these techniques improve speedups compared to vanilla CMT and PMT, they are still limited by the amount of communication they can either accelerate (with special-purpose hardware) or remove (via speculation), as well as by the overheads incurred in facilitating that communication.

### 1.2.1 Software Speculation

Software-based TLS can potentially enable improved parallel performance on today's multicore processors without any hardware changes. Unfortunately, the overhead of tracking dependences and

rollback information in software is prohibitive, thus limiting the type of loops that can be speculated upon and therefore limiting the performance of irregular workloads. We discuss two such software techniques that parallelize loops that are speculatively assumed to be DOALL and that attempt to mitigate the impact of apparent dependences the compiler cannot safely eliminate. While other research has applied software TLS to DOACROSS and DSWP, it does the parallelization and modifies source code manually, in lieu of a fully automatic compiler-based approach [48].

### STMLite

STMLite is a low-cost software transactional memory implementation [41] that improved upon previous implementations by reducing the amount of locking and checking overhead normally encountered with software transactional memory (STM). STMLite accomplishes this by relaxing some restrictions ordinarily encountered with software transactional memories and exploiting the simpler nature of the DOALL loops being parallelized and the fact that only one loop is running at a time. Various benchmarks, including some from SPECfp, were automatically parallelized by selecting loops that the compiler believed (via a profiling pass) to be DOALL, although it couldn't necessarily prove this. Loop iterations were distributed to different threads while speculating that no dependences would be realized at runtime. If it turned out that a dependence *was* realized, STMLite performed a rollback and recovery, re-executing the code in proper sequential order.

Even with this STM implementation optimized for parallel loops, the speedup on the relatively regular SPECfp benchmarks was limited to 2.2×. Ambiguous memory locations accessed in these benchmarks create potential dependences, so they must be placed in a transaction. The authors concluded that STM has limited usefulness unless the number of speculated locations is very small, since the overheads of speculation dwarf any performance improvement.

### DOALL for Clusters

Extending speculative DOALL for a cluster of machines yields impressive, automatically extracted speedups of 43.8× [35]. As with STMLite, loops that appear to be DOALL are parallelized by distributing their iterations to different threads, speculating that few or no dependences will manifest. Without using any speculation, due to the limitations of a static analysis, even for relatively straightforward DOALL loops, speedups are drastically lower, around 4.5×. This highlights the performance impact that apparent dependences can cause, despite never (or rarely) being realized at runtime. However, this boost in speedup is only possible for very regular benchmarks with loops that have few dependences that need to be speculated. Even very small misspeculation rates of <1% can completely dwarf any parallel performance, as misspeculation recovery is very costly. For other benchmarks, the required communication bandwidth to facilitate misspeculation checking greatly limits performance.

## 1.2.2 Hardware Speculation

To overcome the limitations of software TLS, major effort has been devoted to augmenting traditional processors with the required hardware for efficiently parallelizing loops. Some of this hardware accelerates the major functions related to speculation: dependence tracking and misspeculation rollback. This section details some of the most influential efforts to create a compiler–processor co-design that leverages hardware speculation support to speed up automatically parallelized irregular programs.

### HYDRA

The Hydra CMP [25, 26] was one of many early attempts to add dedicated thread-level speculation hardware to a multicore processor. Speedups were obtained by running loop iterations in paral-

lel and speculating that loop-carried dependences would not manifest. Although Hydra primarily targeted loop-level parallelism in the same way as cyclic multithreading, it also targeted function parallelism. Function parallelism was extracted by speculatively assuming that the return value of a function was predictable and that any side effects were not immediately relevant. One thread executed the function, while another continued with code past the function. If an assumption made about the return value or side effects turned out to be incorrect, execution was repeated with the correct ordering.

The custom hardware for speculation performed a number of different functions. Overall, it needed to make sure that a) any thread that read a shared value would receive the most recent value written by any older thread (but not by a younger thread), b) any speculative reads that turned out to be speculated incorrectly would cause execution to rollback, and c) any speculative state would be buffered until it was no longer speculative, at which point it could commit in correct program order. Additional bits were required in the L1 caches to track loads that might be misspeculated and also for facilitating memory renaming, which ensured that threads wouldn't read values written by younger threads. Buffers in the shared L2 cache held any speculative writes until they were safe to retire. These buffers also superseded any accesses to the L2 cache, so that threads could read shared values written from older threads that had not yet committed. Crucial to Hydra's operation was a write-through cache system that allowed all cores to snoop on memory accesses so that they could detect when a misspeculation had occurred and also to ensure, through cache invalidation, that they always loaded the most recently written shared data from older threads alone.

Although loop/function selection was manual, the parallelization process was automatic. Performance improvements were reasonable, ranging from around 1.5× for irregular programs to more than 3× for simpler programs. Later manual SPECint parallelization efforts using Hydra yielded loop/region speedups between 1.24× and 2.1×, depending on the benchmark and the loop/region [47]. The authors noted that the large amount of irregularity in these integer benchmarks

made even manual parallelization difficult. They also acknowledged that some regions that might otherwise be good for parallelization had too few instructions to amortize the TLS overheads, while others had such unpredictable iteration lengths that load imbalances often resulted as short threads stalled waiting for long threads. In general, the irregularity of these benchmarks inhibits successful parallelization.

### STAMPede

Like Hydra, STAMPede [56] is a multicore processor design that seeks to accelerate the performance of automatically parallelized loops by augmenting the hardware with TLS support. As with previous TLS solutions, STAMPede is primarily targeted at loop iteration parallelism in cases where the loop iterations contain mostly apparent dependences. Rather than relying on a write-through cache organization with large speculative buffers, as Hydra does, STAMPede uses a more typical write-back cache scheme with some additional cache coherence messages. Special "epoch" timestamps per thread in addition to extra tracking bits per cache line allow the hardware to detect when an update/invalidation of a speculatively loaded cache line is written by a logically older thread, indicating a misspeculation. By passing around a commit token, threads are able to determine at what point they are no longer speculative and can therefore commit their writes to the rest of the system.

STAMPede explicitly handles instances where a compiler can prove with certainty that an actual dependence always exists and must be communicated, which is often the case for dependences involving registers. Instead of repeatedly misspeculating, as other approaches might do, STAMPede forwards the shared data between threads by placing it in a special region of the stack, and it enforces sequential ordering of thread access with explicit `wait` and `signal` synchronization instructions.

Except for one benchmark, STAMPede only achieves a speedup of 1.21× or lower across a range of irregular benchmarks. A large reason for this low program speedup was that the compiler was unable to find enough promising loops to parallelize, and as a result the overall program coverage

was very low, 28.8% on average.

### POSH

The POSH [38] compiler leverages profiling and a number of different heuristics to extract parallelism on top of assumed TLS hardware. Like other approaches, POSH leverages speculation to remove apparent dependences and thus incurs potential misspeculation costs if it speculates incorrectly. Similarly to STAMPede, it targets not only loop iteration parallelism, but also function-level parallelism. Additionally, POSH exploits parallelism between loops/functions and code immediately following the loops/functions. A profiling run prunes tasks (loops or functions) that are predicted to have low amounts of parallelism—without such profiling, POSH is unable to extract much speedup at all.

Overall, POSH extracts a 1.3× speedup for SPECint 2000 benchmarks. The profiler determines that the most promising loop iterations / function regions in these irregular programs are on the small side—the majority of committed tasks execute fewer than 500 instructions. However, the amount of misspeculation is relatively large—more than 50% of the dynamic tasks are misspeculated and subsequently re-executed. This is not surprising, given the irregular memory access patterns within small loops in SPECint benchmarks. There is something of a silver lining to the large amount of misspeculation: doomed speculative tasks often inadvertently prefetch shared data, which prevents the relatively large load stall that would otherwise happen when the corresponding re-executed task attempted to load it from the L2. Without this effect, POSH's speedups were significantly lower. The observation that loading actual dependences is a potentially large performance bottleneck in part motivates a more careful look at a hardware solution to accelerate communication of these dependences.

### 1.2.3 Custom Architectures

In addition to techniques like TLS, other research has attempted to extract parallelism through even larger overhauls of traditional processor designs. Two such influential projects are Multiscalar [54] and the TRIPS [53] lineage, which includes the TFlex [34] and T3 [51] designs. Through careful compiler–architecture co-design, these projects were able to increase the amount of extracted parallelism at the cost of a larger design effort and more drastic changes to traditional architecture.

#### Multiscalar

Multiscalar processors were designed to extract parallelism from single-threaded programs through aggressive speculative execution on multiple processing units. The associated compiler assigns large blocks of instructions (called *tasks*) to different processing units. A task is an abstracted notion of a contiguous region of dynamic instructions—it can range anywhere from a small number of instructions to an entire basic block, a loop iteration, or an entire function call. These tasks may execute speculatively, with additional hardware used to detect misspeculation and initiate re-execution as needed.

A significant difference from some of the previously discussed TLS techniques is that Multiscalar has a notion of a single logical register file, which accelerates inter-task communication. This register file is physically separated, with a portion of it contained within each processing unit. When the compiler determines that a particular register value may be shared between tasks, it orchestrates the forwarding of the last write to that register from the producer task to the register file of any consuming tasks over a unidirectional ring network. That way, when a consuming task tries to read from that register, if the value has already been produced, it is already available locally (otherwise the hardware will force it to stall). This contrasts with other TLS implementations, which generally rely on reactive memory systems to begin transfer of data when a consumer requests it rather than as soon

as it is produced. This acceleration of inter-task communication of dependences plays an important role in boosting the performance achieved by Multiscalar, as it allows the selection of relatively small tasks [63]. However, this register communication is latency sensitive. On an 8-core design, moving from a 1-cycle latency between cores to a 2-cycle latency decreases Multiscalar performance on some benchmarks by 4–5% [6]. Regrettably, the authors did not sweep communication latency beyond 2 cycles, but this performance drop-off after just one additional cycle of latency serves to highlight how crucial it is to accelerate inter-task dependence communication.

### TRIPS

The general philosophy of the TRIPS line of research is a redesign of conventional ISAs to reshape computation to fit a data-flow model, which can then be used to target different types of parallelism. Instructions directly encode the consumers of the value they produce, by specifying the instruction that will consume the value rather than an explicit register. In one mode of operation, the compiler decomposes execution of a single program into multiple blocks of many instructions each, which are mapped onto available processing elements and executed speculatively. A routing network accelerates the communication of register values between instructions in different blocks, somewhat similarly to Multiscalar. The most recent incarnation of TRIPS achieves good speedups for SPECint 2000, over 3× [51] more than an Intel Atom core, although also with higher energy consumption. Like Multiscalar, TRIPS represents a large architectural redesign, relies heavily on aggressive speculation, and can only accelerate communication of dependences through registers rather than memory. It also shares a similar communication latency sensitivity: when the number of cycles per hop in the routing network was increased from 1 to 2, performance dropped by 20% [22].

## 1.3 Automatic Parallelization of Irregular Programs Must Handle Small Loops

In the previous sections, I presented an overview of significant prior work in the field of automatic parallelization of single-threaded programs, with an emphasis on their success when targeting irregular workloads. A number of general observations follow from this discussion.

Although the basic approaches of cyclic multithreading and pipelined multithreading can be successful for simple programs, they falter in the case of more complex programs. The need to communicate dependences between threads—both actual and apparent—can greatly reduce the amount of extracted parallelism. Thread-level speculation can remove the communication associated with apparent dependences, but the overhead of misspeculating and re-execution greatly limit which loop/code regions can be targeted for parallelization. Small loops, in particular, have too many dependences (often the result of ambiguous memory accesses), and misspeculation happens too frequently to amortize the costs of TLS approaches. However, without targeting small loops / code regions, it is often difficult to find anything worth parallelizing, so program speedup can be limited by low coverage when TLS is applied.

Much of the discussed research acknowledges the importance of accelerating the communication of dependences when possible. To avoid misspeculation penalties, known actual dependences are often handled with explicit communication, either through known memory locations protected by synchronization instructions or through proactive data forwarding with dedicated hardware. Dedicated communication hardware allows Multiscalar and TRIPS to target smaller code regions than other approaches, since data transfer latency for these dependences is reduced. However, many dependences are ambiguous and may or may not manifest, so only a small subset of the dependences can be accelerated. In a similar vein, the authors of POSH observed that fortuitous accidental prefetching of shared data noticeably improved their speedups. These studies highlight the need for

fast communication of dependences, since even with dedicated hardware, program performance is heavily dependent on communication latency.

Overall, this collective body of work points to the reality that in order to achieve good speedup for irregular workloads, the challenging communication demands of small loops must be met head on. However, since speculation cannot be relied upon, due to its overheads, the communication of both apparent and actual dependences must be accelerated. A hardware design capable of facilitating this communication will potentially unlock previously unseen performance for these programs.

### 1.3.1 Hardware Requirements for Parallelizing Small Loops

Here are the properties that hardware support must have if small loops are to be profitably parallelized:

1. Communication must be very fast. Even a slightly higher latency in Multiscalar's register forwarding ring or TRIPS's operand routing network noticeably hurts performance.

2. TLS cannot be used, as the overheads are too high for many loops. Therefore, apparent dependences must also be satisfied/communicated, not just actual dependences.

3. Since all dependences must be handled, dependences involving ambiguous memory references must be accelerated, not only those involving registers. Consequently, a statically unknown amount of shared data, with a statically unknown number of producers and consumers, must be communicated.

In the following chapter, we examine some existing hardware communication mechanisms and explain why they do not meet all the requirements for accelerating small loops. In Chapter 3, we present a novel compiler–architecture co-design that is able to meet all of the challenging properties required for improving the performance of small loops. Our investigation is based on the HELIX

automatic parallelization technique, as this represents the state of the art for compiler-only cyclic-multithreading parallelism. Pipelined multithreading is not as well suited for small loops, due to the difficulty of finding many balanced pipeline stages in small amounts of code (the static size of the control-flow graph limits the size of each pipeline stage).

# 2

# Existing Hardware Cannot Handle Requirements of Small Loops

Chapter 1 detailed prior work on automatic parallelization of irregular programs. The observation that small loops must be targeted to increase the extracted performance of such workloads emerged from that discussion. In particular, the discussion motivated the goal of constructing dedicated hardware to accelerate the communication of inter-thread dependences. This hardware 1) must perform very fast communication, 2) must not rely on thread-level speculation to remove apparent inter-thread dependences but must communicate them instead, and 3) must be able to accelerate the communication of dependences involving memory (as a result of potentially ambiguous pointers),

not just easily detected dependences involving registers—which entails dealing with an unknown number of producers and consumers of shared data.

This chapter will examine existing hardware mechanisms for inter-thread communication and show that none of them fulfill the hardware requirements needed for accelerating communication for irregular programs. First, I discuss cache coherence protocols (the most common inter-thread communication mechanisms) and show them to be inadequate since they transfer data reactively, that is, only when shared data is requested. Next, I explore special-purpose hardware that can facilitate proactive data communication, such as scalar operand networks and software-controlled on-chip networks, but these are still found to be lacking with respect to a number of the requirements. Finally, I examine prior work on custom hardware specifically designed for communicating inter-thread dependences—yet although the custom pieces of hardware are improvements over tradition hardware, they are still lacking. The upshot of this discussion is that all of these existing mechanisms are insufficient for the goal of boosting the performance of irregular programs for HELIX/cyclic-multithreading automatic parallelization, which motivates the creation of the custom compiler–architecture co-design detailed in Chapter 3 to accomplish this task.

## 2.1 Cache Coherence Protocols

The traditional way to communicate data in a commodity multicore processor is through shared memory. Due to the typical presence of per-core private caches, complex cache coherence protocols are necessary to create the simplified illusion of a single cache hierarchy. In reality, the work required to maintain coherence is very complex, to ensure that different copies of the same logical memory location do not exist on the chip. Generally, these protocols transfer data *reactively*. When a core attempts to load a piece of shared data that is not in its local cache, the coherence protocol determines where it is located in the system (either in another core's local cache, in a shared cache, or in main memory) and then fetches it. Depending on the particular computer architecture and cache

coherence implementation, this may involve two or three trips either to the last-level cache (as in some commodity multicore chips) or over an on-chip network to a specific "home" core (as in some many-core chips, such as Tilera [1] or Intel MIC [15])—leading to tens or hundreds of cycles of latency, even with a fast communication fabric. However, these multiple trips are necessary to load the most up-to-date value and to invalidate any outstanding cached data, as dictated by the protocol. Countless different cache coherence protocols have been implemented over the years [55], the exact details of which are well beyond the scope of this dissertation. For our purposes, we will assume a cache coherence protocol typical of Intel's or AMD's commodity multicore processors.

In the case of HELIX (and cyclic multithreading generally), shared data flows between different cores in a cycle, from older loop iterations to younger loop iterations. Within sequential regions of code, a core will often write a piece of shared data. The core running the next loop iteration will then load that piece of shared data when it is safe to do so. Unfortunately, the shared data is almost always located in the previous core's private cache, so it must be transferred locally by the coherence mechanism before it can be used. This creates a constant, pathological communication delay in the already latency-sensitive HELIX style of parallelization.

The following example of a HELIX parallelized loop, shown in Figure 2.1, will serve to highlight this pathological latency. A sequential segment of the loop body reads the shared location X, performs some computation, and then writes the result back to location X. The parallel portion of the loop performs some unrelated, independent calculation. At the beginning of the sequential segment, HELIX inserts a `wait` operation. At the end of each sequential segment, HELIX inserts a `signal` operation. A `wait` operation prevents a core from entering a sequential segment until a corresponding `signal` has been received from the previous iteration of the loop, enforcing the sequential ordering of the sequential segment.

Figure 2.2 shows an execution timeline for a two-core system using this example loop. At the start of execution, core 0 has entered the sequential segment, while core 1 waits to enter. During

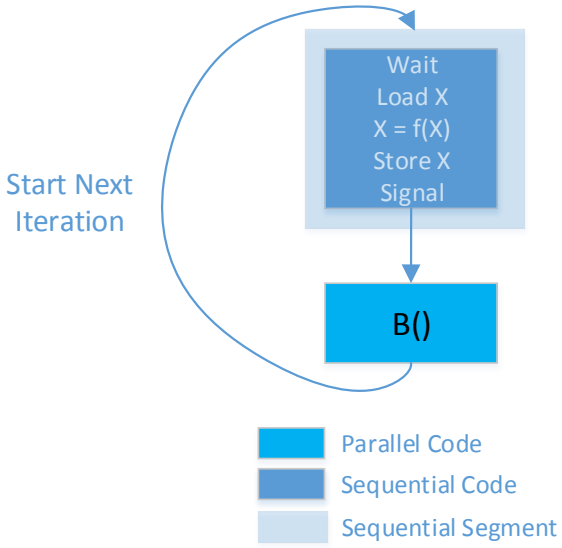



**Figure 2.1:** A loop iteration is decomposed into sequential and parallel portions for HELIX parallelization. The sequential portion accesses a shared memory location that must be communicated between threads.

the execution of the sequential segment, core 0 stores a value to the address of variable X, whose cache line will be loaded into its L1 cache. The core then leaves the sequential segment by issuing a `signal` to unblock core 1. After some communication latency,* core 1 receives the signal and enters the sequential segment. Subsequently, core 1 issues a load to the address of variable X. Because the recently written value of X resides in core 0's L1 cache, there is a cache coherence delay until core 1 receives the data. Since sequential segments are executed in loop iteration order, the data transfer latency significantly increases the critical path execution time.

The cost is a direct result of the *reactive* nature of cache coherence protocols: the data is only moved when it is requested. This produces a coupling effect between the communication of the shared data and the usage of the shared data. Although prefetching of data could alleviate this cou-

*The astute reader may wonder why the `signal` does not suffer the same reactive latency as the data. Depending on the implementation, it might. In the original HELIX paper [12], signals were accelerated through clever prefetching techniques that unfortunately cannot be applied to shared data, due to the unpredictability of their access in irregular programs.

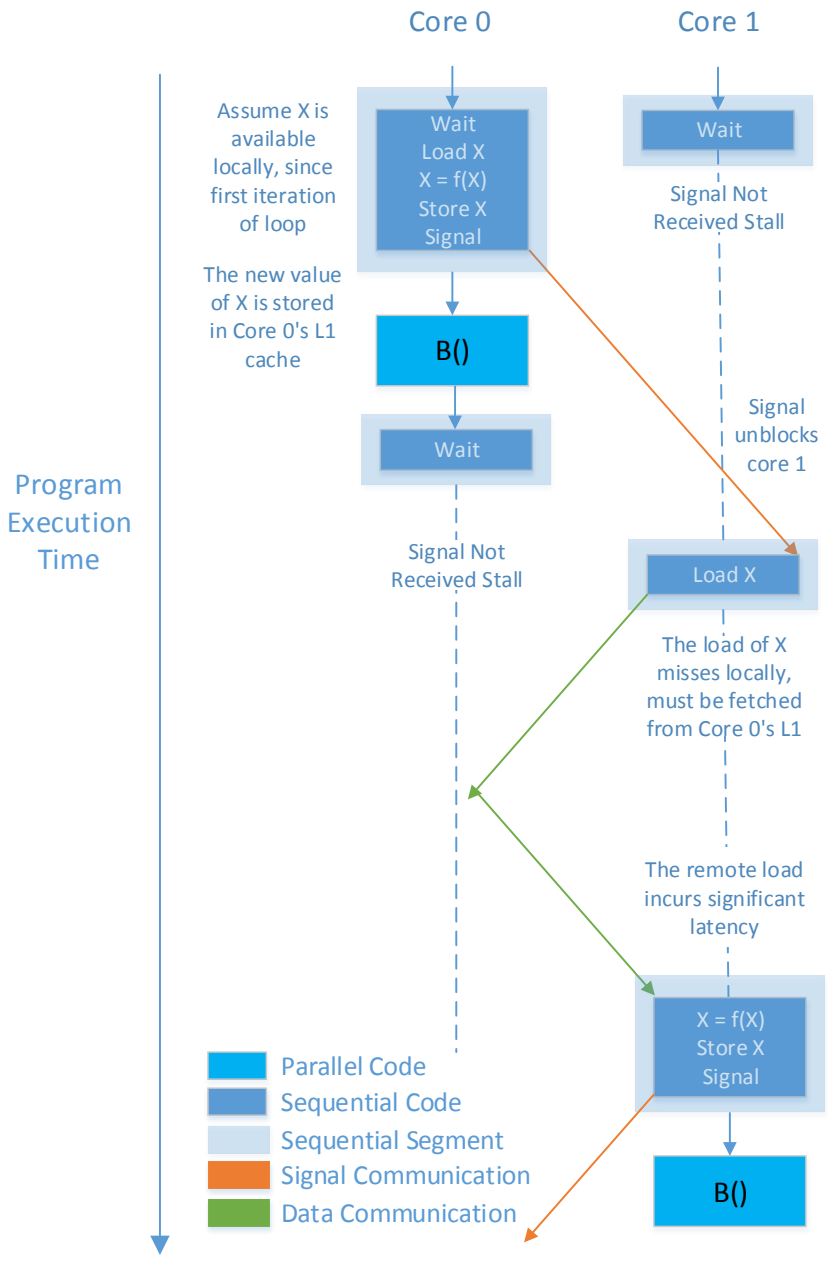



**Figure 2.2:** Typical cache coherence protocols reactively transfer previously written/cached data only when another core attempts to load it. For a two-core system running a HELIX parallelized loop, this leads to a long stall for core 1 as it waits for the data to transfer from core 0's private L1 cache.

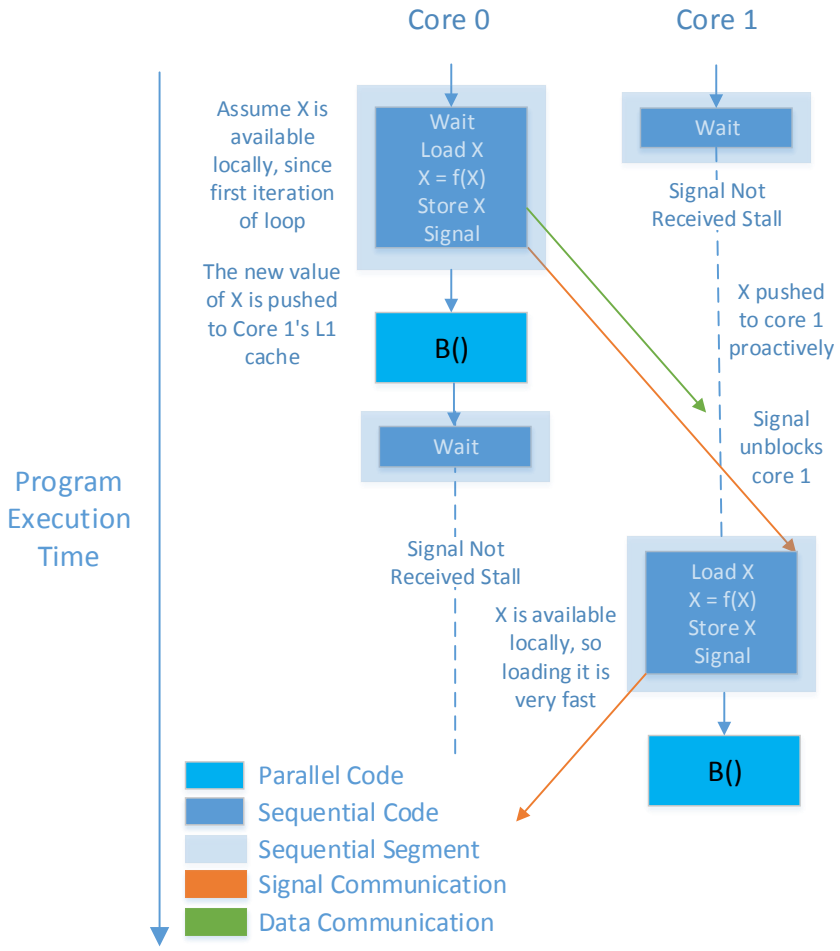



**Figure 2.3:** With a hypothetical proactive data transfer mechanism, core 1 finds the shared data has already arrived when it attempts to fetch it. Core 1 can execute the sequential segment without stalling.

pling effect, the ambiguous pointer memory accesses found within irregular programs are too unpredictable to be reliably prefetched. In contrast, a hypothetical *proactive* communication mechanism would significantly reduce the stall suffered by core 1, as seen in Figure 2.3.

In general, cache coherence protocols target relatively small amounts of data that are shared infrequently between cores, so tens or hundreds of cycles of latency are not a big deal. In contrast, inter-thread communication for small loops requires frequent time-critical data sharing between cores, which makes cache coherence protocols insufficient. A different communication mechanism

is still needed.

## 2.2 Scalar Operand Networks

In contrast to the reactive communication of cache coherence protocols, scalar operand networks are designed to proactively transfer scalar data from the producers of data to the known consumers of that data. Although one can think of the ALU bypass paths in a single processor as an operand network [59], modern scalar operand networks generally connect larger grids of cores [64] or ALUs [21] with a point-to-point network. Because of the streaming/data-flow types of workloads that fit naturally on this type of interconnect, scalar operand networks tend to have very low latency between nodes in the network. The low latency serves to mitigate the cost of operand transport between at least somewhat coupled producers and consumers of data. This section briefly reviews the most notable scalar operand networks and describes why they are not suited for the communication needs of small loops, despite transferring data proactively and with low latency.

### 2.2.1 Tile Processor STN

Processors in the Tile line (n□e RAW processors [58]) consist of many simple cores (e.g., 64) arranged in a grid and connected by several mesh on-chip networks. In some of the Tile processors, one of these networks is a scalar operand network that the designers call the Static Network (STN) [64]. The design goal of this network was to tightly integrate the operand transport network within the core itself, because of the desire for low latency. From the core's point of view, transmission over the STN is accomplished by writing to a specific network-mapped register and receiving from the network is accomplished by reading a specific register. As a result of this tight integration, the latency for reading or writing to the STN is very low.

Routes for particular flows of operands within the STN are set up ahead of time between pairs of cores, in typical circuit-switched fashion. This paradigm favors predictable, long data flows, as in

a streaming model of computation. Circuit-switched routing allows for headerless in-order routing and very low transfer latency for operands between the correct producers and consumers of data, with only one cycle spent for each intermediate router along the route.

Although very effective for a certain class of workloads, even the ultra-fast STN is not well suited for the communication needs of small loops. Because knowledge of the exact producers and consumers of shared data is required, only the simplest provably true register-to-register dependences can be handled by the STN. In addition, ambiguous pointer accesses in small loops make it virtually impossible to determine which threads on which cores will produce or consume any particular piece of data, making the STN useless for these types of dependences.

### 2.2.2 TRIPS OPN

As previously described in Section 1.2.3, the TRIPS custom architecture facilitates an aggressively speculative, data-flow execution model. The TRIPS scalar operand network (which the designers call OPN) interconnects execution, register, and memory resources with a low-latency, point-to-point mesh, single cycle per hop network [22, 64]. Different blocks of instructions run on different execution units, and necessary inter-block register values are routed over the OPN as soon as they are produced. Unlike the routing in the Tile STN, the routing of operands in the TRIPS OPN is dynamic. At runtime, the hardware dynamically routes the operand to wherever the consuming instruction has been assigned.

However, despite the additional flexibility of dynamic routing, the OPN suffers the same critical drawbacks as the STN: the determination of which instructions produce and consume a particular data element must happen in advance, which limits the OPN to handling register-to-register dependences and makes it inapplicable to apparent dependences. Instead, TRIPS handles any apparent/memory dependences with thread-level speculation, which is not usable for small loops. The OPN can hypothetically handle register dependences that are not always communicated, since space is re-

served statically at the consuming execution unit for any value that may arrive, which ensures that the OPN will not back up and stall. However, the need to statically reserve space at the consumer makes the OPN unable to handle dependences based on memory, due to the statically unknown number of locations and consumers.

## 2.3 User-Controlled On-Chip Networks

In addition to or instead of compiler-controlled scalar operand networks, some chips implement user-controllable on-chip networks to support fast, proactive point-to-point communication patterns between processing elements; these include RAW [58] / Tilera [64], the Cell processor [37], and Intel SCC [62]. Many different network topologies have been studied in academia, but simple mesh and ring topologies with simple routing dominate the networks of chips that have seen actual real-world use [30]. These networks were designed to scale arbitrary, proactive inter-core communication beyond the number of cores supportable by a single shared cache. I present two of the most widely used[†] dynamic user-controllable on-chip networks and show that although their increased flexibility makes them better candidates than scalar operand networks for handling the communication demands of parallelized small loops, they are still unsuitable for the task.

### 2.3.1 The Cell Processor Ring Network

The Cell processor [37] combined a high performance core with several simpler in-order co-processors called *synergistic processor elements* (SPEs). The SPEs were specialized for high-throughput floating point and integer arithmetic. Each SPE contained a private scratchpad memory. A very high-bandwidth ring network connected all of the SPEs with the high performance core and the memory controller. Instead of using reactive cache coherence to share data between SPEs, explicit user-controlled direct memory access (DMA) requests transfer data from the SPEs to main memory or

[†]Since these types of on-chip networks are not widely used, this is not saying much.

directly between the private memories of the SPEs.

Unfortunately, the Cell network fails our first requirement of very fast communication. The latency to transfer even a single byte is >100 clock cycles, far higher than desired. Since the network was designed for high-bandwidth bulk transfers of data, it was assumed that the cost of initiating a DMA transfer could be amortized over many bytes.

### 2.3.2 Tile Processor UDN

The Tile Processor architecture contains several on-chip networks, one of which (the UDN) is a point-to-point mesh network dedicated to user-controlled, dynamic message passing between different cores [64]. Unlike some of the other networks in the Tile architecture that facilitate a more traditional reactive cache coherence protocol, the UDN allows direct communication between different threads on different cores. Programmers have the flexibility to send whatever data they choose in whatever fashion they choose, with the only restriction being the hardware resources that are exposed by the architecture. Three different messaging paradigms are supported. Buffered channel and message passing APIs allow flexible, simple usage models with logically unlimited buffering. This makes them easy to use, and programmers do not need to be too concerned about network deadlock. However, their relatively high latency makes them completely unsuitable for the needs of small loops. Raw channels, on the other hand, provide very close access to the hardware, enabling very low-latency sending and receiving of data, at the cost of very limited buffering and the potential need for additional user-defined flow control to avoid deadlock.

The ability to use raw channels to send arbitrary data proactively, with potentially single-digit latencies between cores, provides an interesting option for the communication needs of small loops. These channels would make it relatively easy to communicate register-to-register dependences between known producers and consumers. Since the network transmits packets in order, they could also be safely used to send synchronization signals that could indicate to a consumer core when it is

safe to read a particular dependent value.

However, it is very difficult to see how raw channels could be adapted to handle any dependences—whether they involve memory or registers—when the exact consumers are unknown (or nonexistent). Since consuming cores would not know which core to request produced data from, each core would need to proactively communicate *any* produced data that could potentially be shared to *every* other core in the system—and because the UDN doesn't have any built-in broadcast mechanism, each core would need to explicitly send the dependent data to every other core, creating a huge amount of network traffic. Moreover, in order to avoid deadlock, all the other cores would need to constantly inspect their raw channel output buffers and transfer any received items into local storage in order to prevent the network from deadlocking. Without some hardware mechanism to facilitate the broadcast and automatic storing of data sent through the raw channels, the cores in the system would need to expend a large amount of time merely managing the transmission and reception of this data, even though much of it will never be consumed. All of the work to ensure correct loop-iteration-order sequential access to the shared data would also need to be done by each core through software, an unreasonable task. Although the flexible network fabric seems promising, additional hardware would be needed to handle the required communication demands of small loops.

## 2.4 Other Hardware for Accelerating Communication

Other variants of interconnections have been designed to handle particular communication patterns for automatic parallelization. Previous work on DSWP, for instance, uses a series of relatively simple queues to pipeline dependent data between threads [50]. This so-called synchronization array is limited to cases where there is a one-to-one mapping between the production of values and the consumption of values (i.e., known actual register-to-register dependences), so similarly to scalar operand networks, it is insufficient for our purposes. There is one remaining hardware communication acceleration mechanism that comes close to fulfilling the requirements for small loop commu-

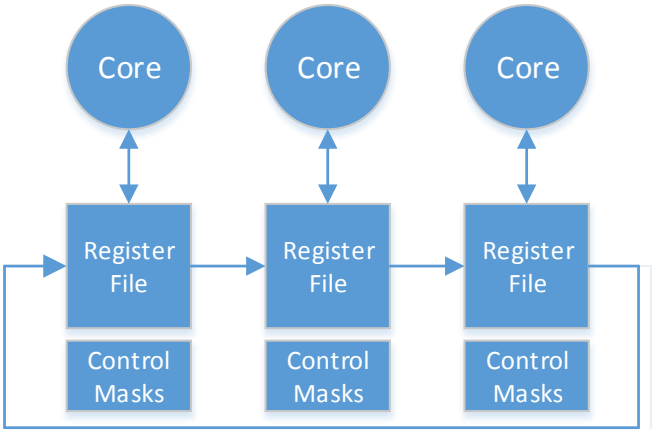



**Figure 2.4:** The Multiscalar distributed register file proactively distributes shared registers around the ring of cores. When a running task attempts to read a shared register, control masks ensure that the value is only accessed if the most recently updated value is present; otherwise, the core stalls until it arrives.

nication and that serves as strong motivation for the hardware solution proposed in Chapter 3.

### 2.4.1 Multiscalar's Distributed Register File

We revisit the distributed register file communication mechanism of the Multiscalar processor that was briefly described in Section 1.2.3, but now in the specific context of accelerating data movement rather than the overall automatic parallelization approach. Recall that the Multiscalar processor enabled automatic parallelization by speculatively executing dynamic blocks of instructions called *tasks* in parallel on multiple cores. When misspeculation occurred (e.g., dependent data was read before it was written), the execution of a task was restarted.

Thread-level speculation helped Multiscalar eliminate the communication cost of apparent inter-task dependences involving memory. To accelerate the communication of register-to-register dependences, a distributed register file was used, as shown in Figure 2.4. Every core contained a portion of this register file, with all of the portions connected by a unidirectional ring network. At compile

time, any task that potentially wrote to registers that were shared amongst tasks would have the corresponding bits of those registers set in one of its per-core control bitmasks. When a shared register value was written by some task, it was automatically forwarded around the ring by the hardware. The register value continued propagating until it updated every core's register file or another task stopped its propagation. A task would stop the propagation of a register if it knew that it might itself write that register, so that younger tasks wouldn't prematurely receive an incorrect value. For a register that would only sometimes be updated by a task (depending on control flow), the task would propagate the unmodified register value as soon as it knew it would not be modifying the value, thus guaranteeing that younger tasks that were expecting an update to that register would remain unblocked.

Through the propagation of control bitmasks from past tasks to future tasks, each task was able to determine which registers would be written by its predecessor tasks and therefore which register values it might need to consume. If a task attempted to read one of these shared register values but it hadn't yet been received, the core would stall until it arrived. If the register *had* already been received, the correct, newly produced value would be present locally, so it could be accessed very quickly. If a task never attempted to read a particular potentially shared register (i.e, if the register dependence was apparent rather than actual), then it would continue execution uninterrupted.

In sum, this distributed register file allowed Multiscalar to accelerate all register communication—both apparent and actual dependences—even when it was unknown whether a task would actually produce or consume a particular register. This nice property was enabled by the proactive forwarding of register values between cores and the intelligent hardware control that orchestrated stalling cores when data was not yet received, stopping propagation of a register if a core might potentially update it, and gracefully handling the case where a register did not actually need to be updated. If a register dependence manifested, the data was made available for consumption only a few cycles after it was produced. If a register dependence did not manifest, cores did not stall unnecessarily.

Unfortunately, despite these nice properties, the Achilles heel of the Multiscalar register file is that it can only be applied to register-to-register dependences, where the registers that may be communicated are statically known. It is impossible to map ambiguous memory accesses to registers, as the addresses and even the number of accessed memory locations is unknown statically. Dynamically mapping memory to registers would be entirely unfeasible, as each core would need to simultaneously perform the same exact mapping. Additionally, the number of shared locations could quickly overwhelm the limited size register file. As such, Multiscalar still relies on thread-level speculation to handle actual and apparent dependences through memory.

## 2.5 Conclusion

In this chapter, we have examined a number of hardware mechanisms for accelerating core-to-core communication. In general, these mechanisms fail the criteria for small loop communication by virtue of 1) having too high a latency, 2) only being able to accelerate statically known register-to-register inter-thread dependences, and/or 3) needing to perform too much communication management in software. Scalar operand networks exemplify property 2), traditional cache coherence protocols exemplify property 1), and user-controlled on-chip networks exemplify properties 2) and 3). On the other hand, the Multiscalar distributed register file comes close to fulfilling the requirements for small loop communication, since it is able to quickly accelerate actual register dependences as well as to gracefully avoid a performance penalty from accelerating apparent or only sometimes seen register dependences. Hardware mechanisms make most of this communication automatic, without intervention from the core. However, the Multiscalar distributed register file fails to accelerate dependences involving memory.

Despite its shortcomings, the Multiscalar register file serves as partial inspiration for the hardware solution I propose in Chapter 3, as it shares a similar ring structure. However, instead of a distributed register file, my HELIX-RC compiler–architecture co-design combines a distributed shared

cache with intelligent synchronization buffering to fully meet the communication demands of small loop parallelization.

# 3

# Automatic Parallelization of Irregular Programs with HELIX-RC

Chapter 1 discussed the drawbacks of existing automatic parallelization techniques with regard to irregular *non-numerical* workloads. Most prior work has attempted to mitigate the high communication demands resulting from parallelizing these workloads by relying on thread-level speculation (TLS). TLS limits the regions of code that can be parallelized. In particular, TLS overheads overwhelm any performance improvement when small loop iterations are parallelized. Thus, TLS cannot be used for small loops. Yet, to extract maximal parallel performance, small loops must be parallelized.

Targeting small loops presents its own set of challenges. Even after extensive code analysis and optimizations, small hot loops will retain actual dependences (in addition to a small number of apparent dependences), typically to share dynamically allocated data. Moreover, since the loop iterations of small loops tend to be short in duration, they require frequent, memory-mediated communication. To run these iterations in parallel, low-latency core-to-core communication is needed for memory traffic. Moreover, ambiguous dependences owing to pointers make it difficult to determine not only the specific shared memory addresses, but also the total amount of shared data. In Chapter 2, we described potential hardware mechanisms that could be used for communication, but found that no existing solution can fulfill the requirements for this kind of communication pattern. The Multiscalar register file was the closest, as it was able to proactively communicate both actual and apparent register-to-register dependences; however, it was unable to accelerate memory dependences without relying on TLS.

To meet the communication demands for short loops, we present HELIX-RC, a co-designed architecture–compiler parallelization framework for chip multiprocessors. The compiler identifies which data must be shared between cores, and the architecture proactively circulates this data along with synchronization signals among the cores rather than waiting for a request. The proactive communication immediately circulates shared data as early as possible—thus decoupling communication from computation. HELIX-RC builds on the HCCv1 compiler, developed for the first iteration of HELIX [12, 13], which automatically generates parallel code for commodity multicore processors. Because performance improvements from HCCv1 saturate at four cores, due to communication latency, I propose *ring cache* as an architectural enhancement that facilitates low-latency core-to-core communication to satisfy inter-thread memory dependences, relying on guarantees provided by the co-designed *HCCv3* compiler to keep it lightweight.

HELIX-RC automatically parallelizes irregular programs with unmatched performance improvements. Across a range of SPECint 2000 benchmarks, decoupling communication and computation

enables a threefold improvement in performance over HCCv1, on a simulated multicore processor consisting of 16 Atom-like, *in-order* cores with a ring cache that has 1KB of memory per node (32× smaller than the L1 data cache). The proposed system offers an average speedup of 6.85× over unparallelized code running on a single core. Detailed evaluations show that even with a conservative ring cache configuration, HELIX-RC is able to achieve 95% of the possible speedup with unlimited resources (i.e., unbounded bandwidth, instantaneous intercore communication, and unconstrained size).

The remainder of this chapter further describes the motivation for HELIX-RC and the results of implementing it. I first review the limitations of compiler-only improvements and identify co-design opportunities for improving the thread-level parallelism (TLP) of loop iterations. Next, I explore the speedups obtained by decoupling communication from computation with compiler support. After describing the overall HELIX-RC approach, I delve more deeply into both the compiler and the hardware enhancement. Finally, I use a detailed simulation framework to evaluate the performance of HELIX-RC and analyze its sensitivity to architectural parameters.

## 3.1 Background and Opportunities

### 3.1.1 Limits of Compiler-only Improvements

To understand what limits the performance of parallel code extracted from irregular programs, I began with HCCv1 [12, 13], a state-of-the-art parallelizing compiler.

HCCv1 This first-generation compiler automatically generates parallel threads from sequential programs by distributing successive loop iterations across adjacent cores within a single multicore processor, similar to conventional DOACROSS parallelism [16]. Since there are data dependences between loop iterations (i.e., *loop-carried dependences*), some segments of a loop's body—called *sequential segments*—must execute in iteration order on the separate cores to preserve the semantics

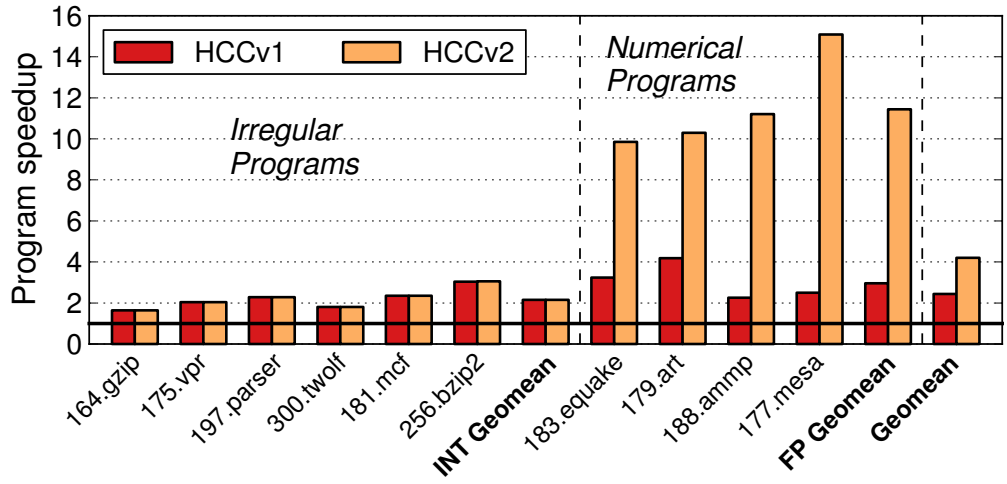



**Figure 3.1:** Improving the HCCv1 compiler alone does not improve performance for SPECint 2000 benchmarks.

of the sequential code. Synchronization operations mark the beginning and end of each sequential segment.

HCCv1 includes a large set of code optimizations (e.g., code scheduling, method inlining, loop unrolling), most of which are specifically tuned to extract TLP. Despite this, performance improvements obtained by the original HCCv1 compiler saturate at four cores, due to high core-to-core communication latency.

HCCv2 I first improved the code analysis and transformation. Specifically, I increased the accuracy of both data dependence and induction variable analysis, and I added other transformations to extract more parallelism (e.g., scalar expansion, scalar renaming, parallel reductions, and loop splitting [2]). I call this improved compiler *HCCv2*.

Figure 3.1 compares speedups for HCCv1 and HCCv2 based on simulations of parallel code generated by each when targeting a 16-core processor with an optimistic 10-cycle core-to-core communication latency.* The engineering improvements of HCCv2 significantly increased speedups over

*Details of this experiment are presented in Section 3.5.

HCCv1 for numerical programs (SPECfp 2000), from 2.4× to 11×. HCCv2 successfully parallelized the numerical programs because the data dependence analysis is highly accurate for loops at almost any level of the loop nesting hierarchy. Furthermore, the improved compiler removed the remaining actual dependences among registers (e.g., via parallel reduction) to generate loops with long iterations that can run in parallel on different cores.

Unfortunately, irregular programs (SPECint) are not as compliant to the compiler improvements and saw little to no benefit from HCCv2. Because core-to-core communication in conventional systems is expensive, the compiler must parallelize large loops (the larger the loop with loop-carried dependences, the less frequently cores synchronize), which limits the accuracy of the dependence analysis and thereby limits TLP extraction. This is why HELIX-RC focuses on small (hot) loops to parallelize this class of programs. My hypothesis is that modest architectural enhancements co-designed with a compiler that targets small loops can successfully parallelize irregular programs.

### 3.1.2 Opportunity

There is an opportunity to aggressively parallelize irregular programs based on the following insights: (i) small loops are easier to analyze with high accuracy, (ii) predictable computation means that most of the required communication updates shared memory locations, (iii) we can efficiently satisfy the communication demands of actual dependences for small loops with low-latency, core-to-core communication, and (iv) proactive communication efficiently hides communication latencies.

Accurate data dependence analysis is possible for small loops. The accuracy of data dependence analysis increases for smaller loops because (i) there is less code—and therefore less complexity—to analyze, and (ii) the number of possible aliases for a pointer in the code scales down with code size. In other words, we can avoid the conservative pointer aliasing assumptions that lower accuracy for large loops.

To evaluate the accuracy of the data dependence analysis for small loops using modern compilers, I started with a state-of-the-art analysis called VLLPA [23]. Figure 3.2 shows that the initial accuracy of this analysis (i.e., the average number of actual data dependences compared to all dependences identified for the set of loops HELIX-RC ended up selecting for parallelization) was 48%. To improve the accuracy, I extended VLLPA (i) to be fully flow-sensitive [14], i.e., to track the values of both registers and memory locations according to their position in the code, (ii) to be path-based, i.e., to name runtime locations according to how they are accessed from program variables [19], (iii) to exploit data type and type casting information to conservatively eliminate incompatible aliases, and (iv) to exploit standard library-call semantics. Figure 3.2 shows that these extensions increased the accuracy of the analysis for small loops to 81%. As a result, most of the loop-carried data dependences identified by the compiler are actual and therefore require core-to-core communication. The remaining 19% can always be handled by speculation.

Most required communication is for updating shared memory locations. Sharing data among loop iterations requires core-to-core communication to propagate new values when loop iterations run on different cores. However, if new values are predictable (e.g., incrementing a shared variable at every iteration), communication can be avoided. I extended the variable analysis in HCCv1 to capture the following predictable variables: (i) induction variables for which the update function is a polynomial up to the second order, (ii) accumulative, maximum, and minimum variables, (iii) variables set but not used until after the loop, and (iv) variables set in every iteration, even when the updated value is not constant. If a variable falls into any of these categories, each core can independently recompute its correct value.

Exploiting the predictability of variables—again for small loops in irregular programs—allows the compiler to remove a large fraction of the communication required to share registers. Figure 3.4 compares a naive solution that propagates new values for all loop-carried data dependences (100%)

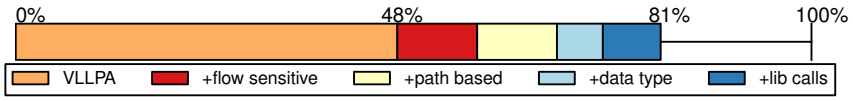



**Figure 3.2:** Various improvements to the VLLPA data dependence analysis boost accuracy significantly for small hot loops in SPECint 2000.

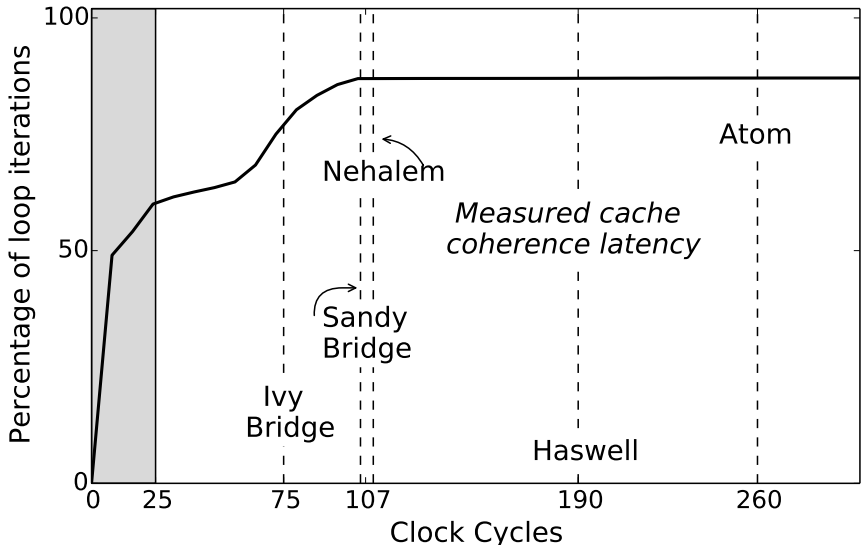



**Figure 3.3:** The most promising loops to parallelize (as determined by the compiler) usually have very short iterations, especially compared to a typical cache coherence latency.

versus a solution that exploits variable predictability. By recomputing variables, the majority of the remaining communication is for shared memory locations rather than registers.

Communication for small hot loops must be fast. While the simplicity of small loops allows for easy analysis, small loops have short iterations—typically less than 100 clock cycles. Because these short iterations require (at least) some communication to run in parallel, efficient parallel execution demands a low-latency core-to-core communication mechanism.

To better understand this need for fast communication, Figure 3.3 plots a cumulative distribution of average iteration execution times on a single Atom-like core (described in Section 3.5) for the set

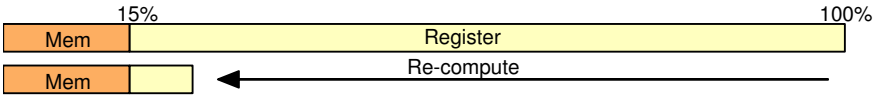



**Figure 3.4:** Predictability of variables reduces register communication. The remaining required communication is mostly through memory.

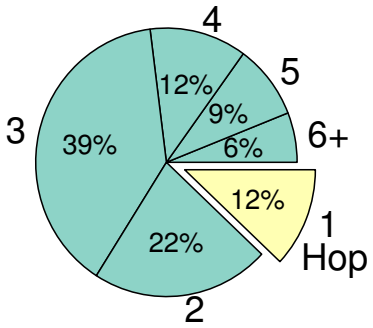



**Figure 3.5:** Most required data transfers for small loops are between non-adjacent cores in a hypothetical 16-core system connected by a ring network.

of hot loops from SPECint 2000 benchmarks chosen for parallelization by HELIX-RC. The shaded portion of the plot shows that more than half of the loop iterations complete within 25 clock cycles. The plot also delineates the measured core-to-core round trip communication latencies for three modern multicore processors. Even for the shortest-latency machine, Ivy Bridge, 75 cycles is much too long for the majority of these short loops. Of course, a conventional region-extending transformation such as loop unrolling could lengthen the duration of these inner loops, but this would also increase the lengths of sequential segments, reducing exploitable parallelism.

Proactive communication achieves low latency by decoupling communication from computation. A compiler must conservatively assume that dependences exist between all iterations for most of the loop-carried dependences in irregular programs. Because of the complexity of the control and data flows in such programs, a compiler cannot easily infer the distance between

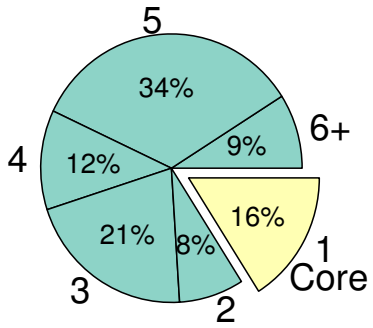



**Figure 3.6:** Most shared data for small loops is consumed by multiple cores in a hypothetical 16-core system.

a loop iteration that generates data and the ones that consume it. For conventional synchronization approaches [12, 44, 45, 60, 65, 66], this assumption of dependences between all subsequent iterations leads to *sequential chains* that severely limit the performance sought by running loop iterations in parallel.[†] These sequential chains, which include both communication and computation, have two sources of inefficiency. First, adjacent-core synchronization often turns out *not* to be necessary for every link of these chains. Second, when data forwarding is initiated lazily (at request time), it blocks computation while waiting for data transfers between cores.

Finally, for loops parallelized by HELIX-RC, most communication is not between successive loop iterations. Hence, because HELIX-RC distributes successive loop iterations to adjacent cores, most communication is not between adjacent cores. Figure 3.5 charts the distribution of undirected distances between data-producing cores and their first consumer core on a platform with 16 cores organized in a ring. Only 12% of those transfers are between adjacent cores. Moreover, Figure 3.6 shows that most (84%) of the shared values from these loops are consumed by multiple cores. Since consumers of shared values are not known at compile time, HELIX-RC implements a mechanism that proactively broadcasts data and signals to all other cores. Such proactive communication, which

[†]Others have called this chain a *critical forwarding path* [57, 66].

does not block computation, is the cornerstone of the HELIX-RC approach.

## 3.2 The HELIX-RC Solution

The goal of HELIX-RC is to decouple *all* communication required to efficiently run iterations of small hot loops in parallel. This is realized by *decoupling value forwarding from value generation* and by *decoupling signal transmission from synchronization*. I will now explain how HELIX-RC achieves such decoupling.

### 3.2.1 Approach

HELIX-RC is a co-design of compiler (*HCCv3*) and architectural (*ring cache*) enhancements. HCCv3 distinguishes parallel code (i.e., code outside any sequential segment) from sequential code (i.e., code within sequential segments) by using two instructions that extend the instruction set. The ring cache is a ring network that connects *ring nodes* attached to each core in the processor to operate during sequential segments as a distributed first-level cache that precedes the private L1 caches. The hardware support can be simple and efficient because it relies on compiler-guaranteed properties of the code. The following paragraphs summarize the main components of HELIX-RC.

ISA A pair of instructions—`wait` and `signal`—are introduced to mark the beginning and end of a sequential segment. Each of these instructions has an integer value as a parameter that identifies the particular sequential segment. The `wait` instruction blocks execution of the core that issued it (e.g., `wait 3`) until all other cores have finished executing the corresponding sequential segment, which they signify by executing the appropriate `signal` instruction (e.g., `signal 3`).

Compiler HCCv3 takes sequential programs and parallelizes loops that are most likely to speed up performance when their iterations execute in parallel. Only one loop at a time runs in parallel, and its successive iterations run on cores organized as a unidirectional ring.

To satisfy loop-carried data dependences, HCCv3 keeps the execution of sequential segments in iteration order by inserting `wait` and `signal` instructions to delimit the entry and exit points of these segments. In this way, HCCv3 guarantees that accesses to a variable or another memory location that might need to be shared between cores are always within sequential segments. Moreover, shared variables (normally allocated to registers in sequential code) are mapped to specially allocated memory locations. Hence, accesses to these variables within sequential segments occur via memory operations.

Core A core forwards all memory accesses *within* sequential segments to its local ring node. All other memory accesses (not within a sequential segment) go through its private L1 cache. To determine whether the executing code is part of a sequential segment, a core simply counts the number of executed `wait` and `signal` instructions. If more `wait`s have been executed than matching `signal`s, then the executing code belongs to a sequential segment.

Memory The ring cache is a connected ring of nodes, one per core. Each ring node has a cache array that satisfies both loads and stores received from its attached core.

HELIX-RC does not require other changes to the existing memory hierarchy because the ring cache orchestrates interactions with it. To avoid any changes to conventional cache coherence protocols, the ring cache permanently maps each memory address to a unique ring node. All accesses from the distributed ring cache to the next cache level (L1) go through the associated node for a corresponding address.

### 3.2.2 Decoupling Communication From Computation

Having introduced the main components of HELIX-RC, we now describe how they interact to efficiently decouple communication from computation.

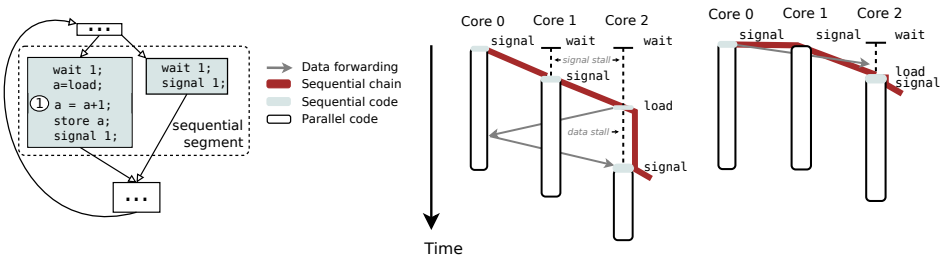
Shared data communication HELIX-RC decouples communication of variables and other shared data locations from computation by propagating new shared data through the ring cache as soon as it is generated. Once a ring node receives a store, it records the new value and proactively forwards its address and value to an adjacent node in the ring cache, all without interrupting the execution of the attached core. The value then propagates from node to node through the rest of the ring without interrupting the computation of any core—thus *decoupling communication from computation*.

Synchronization Given the difficulty of determining which iteration depends on which in irregular programs, compilers typically make the conservative assumption that an iteration depends on all of its predecessor iterations. Therefore, a core cannot execute sequential code until it is unblocked by its predecessor [12, 44, 57]. Moreover, an iteration unblocks its successor only if both it and its predecessors have executed this sequential segment or they are not going to. This execution model leads to a chain of signal propagation across loop iterations that includes unnecessary synchronization: even if an iteration is not going to execute sequential code, it still needs to synchronize with its predecessor before unblocking its successor.

HELIX-RC removes these synchronization overheads by enabling an iteration to detect the readiness of all predecessor iterations, not just one. Therefore, once an iteration forgoes executing the sequential segment, it immediately notifies its successor without waiting for its predecessor. Unfortunately, while HELIX-RC removes unnecessary synchronization, it increases the number of signals that can simultaneously be in flight.

HELIX-RC relies on the new `signal` instruction to handle synchronization signals efficiently. Synchronization between a producer and a consumer involves (i) the producer generating a signal, (ii) the consumer requesting that signal, and (iii) signal transmission between the two.

On a conventional multicore processor, which relies on a pull-based memory hierarchy for com-




**(a)** Parallel code **(b)** Coupled communication **(c)** Decoupled communication

**Figure 3.7:** Example illustrating benefits of decoupling communication from computation. On the left, a representative loop from 175.vpr toggles between a necessary/unnecessary sequential segment. If communication is coupled, signal and data stalls slow down computation.

munication, signal transmission is inherently lazy, and signal requests and transmissions get serialized. In contrast, in HELIX-RC, `signal` instructs the ring cache to proactively forward a signal to all other nodes in the ring without interrupting any of the cores, thereby *decoupling signal transmission from synchronization*.

Code example Given the importance of these decoupling mechanisms to fully realize performance benefits, let's explore how HELIX-RC implements them using a concrete example. The code in Figure 3.7(a), abstracted for clarity, represents a small hot loop from 175.vpr of SPECint 2000 that is responsible for 55% of that program's total execution time. The loop contains a sequential segment with two possible execution paths. The left path contains an actual dependence in which instances of instruction `1` in an iteration use values from previous iterations. The right path does not depend on prior data. Because the compiler cannot predict the execution path of a particular iteration (due to complex control flow), it must assume that in any given iteration, instruction `1` depends on the previous iteration. Therefore, it must synchronize all successive iterations by inserting `wait` and `signal` instructions on every execution path. Figure 3.7(b) highlights this sequential chain in red. Now assume that only iterations 0 and 2, running on cores 0 and 2, respectively, exe-

cute instruction 1. In this case, the sequential chain is unnecessarily long because of the superfluous `wait` in iteration 1. Each iteration waits (via the `wait` instruction) for the signal generated by the `signal` instruction of the previous iteration. Also, iterations that update `a` (iterations 0 and 2) must load previous values first (using a regular load). Hence, two sets of stalls slow down the chain. First, iteration 1 performs unnecessary synchronization (signal stalls), because it only contains parallel code. Second, lazy forwarding of the shared data leads to data stalls, because the transfer only begins when requested, at a load, rather than when generated, at a store.

HELIX-RC proactively communicates data and synchronization signals between cores, which leads to the more efficient scenario shown in Figure 3.7(c). The sequential chain now includes only the delay required to satisfy the dependence, that is, communication updating a shared value. As a side note, a TLS-based solution suffers in this scenario. Only a complex speculation approach might be able to capture the run-time behavior of this dependence (i.e., speculating when the program will execute which path), because 175.vpr frequently toggles between the two paths. Moreover, speculation cannot avoid the data-forwarding overhead when the program executes the branch with instruction 1, adding communication delay to the already critical sequential chain.

## 3.3 Compiler

The decoupled execution model of HELIX-RC described so far is possible given the tight co-design of the compiler and architecture. In this section, we focus on compiler-guaranteed code properties that enable a lightweight ring cache design, and we follow up with code optimizations that make use of the ring cache.

### Guaranteed code properties

- Only one loop at a time can run in parallel. Apart from a dedicated core responsible for executing code outside parallel loops, each core is either executing an iteration of the current

loop or waiting for the start of the next one.

- Successive loop iterations are distributed to threads in a round-robin manner. Since each thread is pinned to a predefined core and cores are organized in a unidirectional ring, successive iterations form a logical ring.
- Communication between cores executing a parallelized loop occurs only within sequential segments.
- Different sequential segments always access different shared data. HCCv3 only generates multiple sequential segments when there is no intersection of shared data. Consequently, instances of distinct sequential segments may run in parallel.
- At most two `signal`s per sequential segment emitted by a given core can be in flight at any time. Hence, only two `signal`s per segment need to be tracked by the ring cache.

This last property eliminates unnecessary `wait` instructions while keeping the architectural enhancement simple. Eliminating `wait`s allows a core to execute a later loop iteration than its successor (significantly boosting parallelism). Future iterations, however, produce `signal`s that must be buffered. The last code property prevents a core from getting more than one "lap" ahead of its successor. Therefore, when buffering signals, each ring cache node only needs to recognize two types—those from the past and those from the future.

Code optimizations In addition to the optimizations of HCCv2, HCCv3 includes optimizations that are essential for the best performance of irregular programs on a ring-cache-enhanced architecture: aggressive splitting of sequential segments into smaller code blocks, identification and selection of small hot loops, and elimination of unnecessary `wait` instructions.

Sizing sequential segments poses a tradeoff. Additional segments created by splitting can run in parallel with others, but extra segments entail extra synchronization, which adds communication

overhead. Thanks to decoupling, HCCv3 can split more aggressively than HCCv2 to significantly increase TLP. Note that segments cannot be split indefinitely—each shared location must belong to only one segment.

To identify small hot loops that are most likely to speed up when their iterations run in parallel, HCCv3 includes a profiler to capture the behavior of the ring cache. Whereas HCCv1 relies on an analytical performance model to select the loops to parallelize, HCCv3 profiles loops on representative inputs. During profiling, instrumentation code emulates execution with the ring cache, resulting in an estimate of the time saved by parallelization. Finally, HCCv3 uses a loop-nesting graph, annotated with the profiling results, to choose the most promising loops.

## 3.4 Architecture Enhancements

Adding a ring cache to a multicore architecture enables the proactive circulation of data and signals thats boost parallelization. This section describes the design of the ring cache and its constituent ring nodes. The design is guided by the following objectives:

Low-latency communication HELIX-RC relies on fast communication between cores in a multicore processor for synchronization and for data sharing between loop iterations. Since low-latency communication is possible between physically adjacent cores in modern processors, the ring cache implements a simple unidirectional ring network.

Caching shared values A compiler cannot easily guarantee whether and when shared data generated by a loop iteration will be consumed by other cores running subsequent iterations. Hence, the ring cache must cache shared data. Keeping shared data on local ring nodes provides quick access for the associated cores. As with data, it is also important to buffer signals in each ring node for immediate consumption.

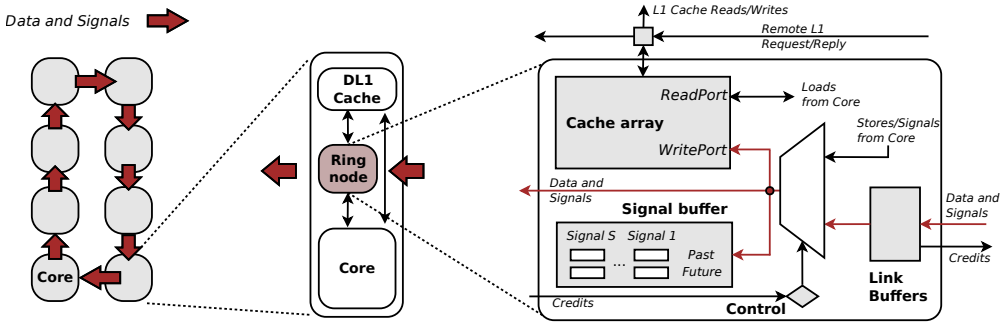



**Figure 3.8:** Ring cache architecture overview. From left to right: overall system; single core slice; ring node internal structure.

EASY INTEGRATION The ring cache is a minimally invasive extension to existing multicore systems, easy to adopt and integrate. It does not require modifications to the existing memory hierarchy or to cache coherence protocols.

With these objectives in mind, we now describe the internals of the ring cache and its interaction with the rest of the architecture.

### 3.4.1 RING CACHE ARCHITECTURE

The ring cache architecture relies on the following properties of the compiled code: (i) parallelized loop iterations execute in separate threads on separate cores, arranged in a logical ring, and (ii) data shared between iterations moves between cores from current to future iterations. These properties imply that the data involved in timing-critical dependences that potentially limit overall performance are both produced and consumed in the same order as loop iterations. Furthermore, a ring network topology captures this data flow, as sketched in Figure 3.8. The following paragraphs describe the structure and purpose of each ring cache component.

RING NODE STRUCTURE The internal structure of a per-core ring node is shown in the right half of Figure 3.8. Parts of this structure resemble a simple network router. Unidirectional links connect a node to its two neighbors to form the ring backbone. Bidirectional connections to the core and

the private L1 cache allow injection of data into and extraction of data from the ring. There are three separate sets of data links and buffers. A primary set forwards data and signals between cores. Two other sets manage infrequent traffic for integration with the rest of the memory hierarchy (see Section 3.4.2). Separating these three traffic types simplifies the design and avoids deadlock. Finally, signals move in lockstep with forwarded data to ensure that a shared memory location is not accessed before the data arrives.

In addition to these router-like elements, a ring node also contains structures more common to caches. A set-associative *cache array* stores all data values (and their tags) received by the ring node, whether from a predecessor node or from its associated core. The line size of this cache array is kept at one machine word. While the small line is contrary to typical cache designs, it ensures there will be no false data sharing by independent values from the same line.

The final structural component of the ring node is the *signal buffer*, which stores signals until they are consumed.

Node-to-node connection The main purpose of the ring cache is to proactively provide many-to-many core communication in a scalable and low-latency manner. In the unidirectional ring formed by the ring nodes, data propagates by *value circulation*. Once a ring node receives an (address, value) pair, either from its predecessor or from its associated core, it stores a local copy in its cache array and propagates the same pair to its successor node. The pair eventually propagates through the entire ring (stopping after a full cycle), so that any core can consume the data value from its local ring node, as needed.

This value circulation mechanism allows the ring cache to communicate between cores more quickly than reactive systems (such as most coherent cache hierarchies). In a reactive system, data transfer only begins when the receiver requests the shared data, which adds transfer latency to an already latency-critical code path. In contrast, a proactive scheme overlaps transfer latencies with

computation to lower the receiver's perceived latency.

The ring cache prioritizes the common case where data generated within sequential segments must propagate to all other nodes as quickly as possible. Assuming no contention over the network and single-cycle node-to-node latency, the design shown in Figure 3.8 allows us to bound the latency for a full trip around the ring to $N$ clock cycles, where $N$ is the number of cores. Each ring node prioritizes data received from the ring and stalls injection from its local core.

In order to eliminate buffering delays within the nodes that are not due to L1 traffic, the number of write ports in each node's cache array must match the link bandwidth between two nodes. While this may seem like an onerous design constraint for the cache array, Section 3.5.3 shows that just one write port is sufficient to reap more than 99% of the ideal-case benefits.

To ensure correctness under network contention, the ring cache is sometimes forced to stall all messages (data and signals) traveling along the ring. The only events that can cause contention and stalls are ring cache misses and evictions, which may then necessitate fetching data from a remote L1 cache. While these ring stalls are necessary to guarantee correctness, they are infrequent.

The ring cache relies on credit-based flow control [30] and is deadlock free. Each ring node has at least two buffers attached to the incoming links to guarantee forward progress. The network maintains the invariant that there is always at least one empty buffer somewhere in the ring per set of links. That is why a node only injects new data from its associated core into the ring when there is no data from a predecessor node to forward.

Node–core integration Ring nodes are connected to their respective cores as the closest level in the cache hierarchy (Figure 3.8). The core's interface to the ring cache is through regular loads and stores for memory accesses in sequential segments.

As previously discussed, `wait` and `signal` instructions delineate code within a sequential segment. A thread that needs to enter a sequential segment first executes a `wait`, which is only returned

by the associated ring node when matching signals have been received from all other cores executing prior loop iterations. The signal buffer within the ring node enforces this. Specialized core logic detects the start of the sequential segment and routes memory operations to the ring cache.[‡] Finally, execution of the corresponding `signal` marks the end of the sequential segment.

The `wait` and `signal` instructions require special treatment in out-of-order cores. Since they may have system-wide side effects, these instructions must issue non-speculatively from the core's store queue, and regular loads and stores cannot be reordered around them. My implementation reuses logic from load–store queues for memory disambiguation and holds a lightweight local fence in the load queue until the `wait` returns to the senior store queue. This is not a concern for in-order cores.

### 3.4.2 Memory Hierarchy Integration

The ring cache is a level within the cache hierarchy and as such must not break any consistency guarantees that the hierarchy normally provides. Consistency between the ring cache and the conventional memory hierarchy results from the following invariants: (i) shared memory can only be accessed within sequential segments through the ring cache (compiler enforced) (ii) only a uniquely assigned *owner* node can read or write a particular shared memory location through the L1 cache on a ring cache miss (ring cache enforced) and (iii) the cache coherence protocol preserves the order of stores to a memory location through a particular L1 cache.[§]

Sequential consistency To preserve the semantics of a parallelized single-threaded program, memory operations on shared values require sequential consistency. The ring cache meets this requirement by leveraging the unidirectional data flow guaranteed by the compiler. Sequential consis-

[‡]This feature may add one multiplexer delay to the critical delay path from the core to the L1 cache.

[§]Most cache coherence protocols (including Intel, AMD, and ARM implementations) provide this minimum guarantee.

tency must be preserved when ring cache values reach lower-level caches, but the consistency model provided by conventional memory hierarchies is weaker. I resolve this difference by introducing a single serialization point per memory location, namely, a unique *owner* node responsible for all interactions with the rest of the memory hierarchy. When a shared value is moved between the ring cache and L1 caches (owing to occasional ring cache load misses and evictions), only its owner node can perform the required L1 cache accesses. This solution preserves existing consistency models with minimal impact on performance.

Cache flush Finally, to guarantee coherence between parallelized loops and serial code between loop invocations, each ring node flushes the dirty values of memory locations it owns to its core's L1 once a parallel loop has finished execution. This is equivalent to executing a distributed fence at the end of loops. In a multiprogram scenario, signal buffers must also be flushed/restored at program context switches.

## 3.5 Evaluation

As a result of the compiler being co-designed with the architecture, HELIX-RC more than triples the performance of parallelized code compared to a compiler-only solution (i.e., HCCv2). This section investigates HELIX-RC's performance benefits and their sensitivity to ring cache parameters. I confirm that the majority of speedups come from decoupling all types of communication and synchronization. I conclude by analyzing the execution model's remaining overheads.

### 3.5.1 Experimental Setup

I ran experiments on two sets of architectures. The first relies on a conventional memory hierarchy to share data among the cores. The second relies on the ring cache.

**Table 3.1:** Characteristics of parallelized benchmarks.

| Benchmark | Phases | Parallel loop coverage | | |
|---|---|---|---|---|
| | | HELIX-RC | HCCv2 | HCCv1 |
| Integer benchmarks | | | | |
| 164.gzip | 12 | 98.2% | 42.3% | 42.3% |
| 175.vpr | 28 | 99% | 55.1% | 55.1% |
| 197.parser | 19 | 98.7% | 60.2% | 60.2% |
| 300.twolf | 18 | 99% | 62.4% | 62.4% |
| 181.mcf | 19 | 99% | 65.3% | 65.3% |
| 256.bzip2 | 23 | 99% | 72.3% | 72.1% |
| Floating point benchmarks | | | | |
| 183.equake | 7 | 99% | 99% | 77.1% |
| 179.art | 11 | 99% | 99% | 84.1% |
| 188.ammp | 23 | 99% | 99% | 60.2% |
| 177.mesa | 8 | 99% | 99% | 64.3% |

Simulated conventional hardware Unless otherwise noted, I simulated a multicore in-order x86 processor by adding multiple-core support to the XIOSim simulator. The single-core XIOSim models have been extensively validated against an Intel® Atom processor [32]. I used XIOSim because it is a publicly available simulator that is able to simulate fine-grained microarchitectural events with high precision.

The simulated cache hierarchy has two levels: a per-core 32KB, 8-way associative L1 cache and a shared 8MB 16-bank L2 cache. I varied the core count from 1 to 16, but did not vary the amount of L2 cache with the number of cores, keeping it at 8MB for all configurations. Scaling the cache size would have made it difficult to distinguish the benefits of parallelizing a workload from the benefits of fitting its working set into the larger cache, causing misleading results. Finally, I used DRAMSim2 [52] for cycle-accurate simulation of memory controllers and DRAM.

I extended XIOSim with a cache coherence protocol that assumes an optimistic cache-to-cache latency of 10 clock cycles. This 10-cycle latency is optimistically low even compared to research pro-

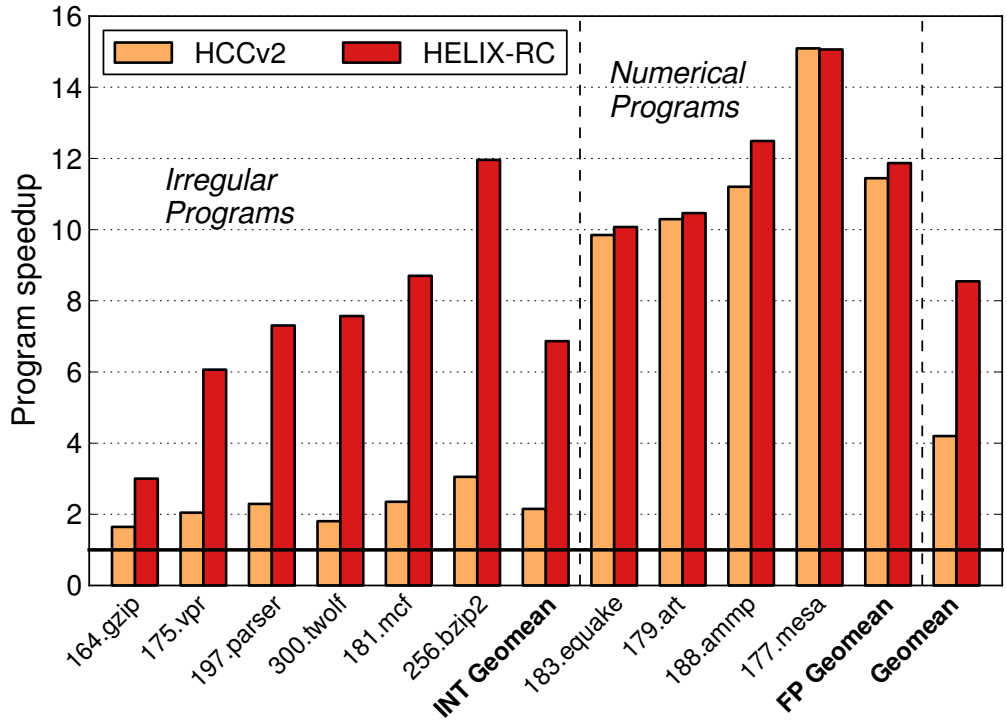



**Figure 3.9:** HELIX-RC triples the speedup obtained by HCCv2. Speedups are relative to sequential program execution. Because the integer benchmarks had much more communication that needed to be accelerated, they were helped much more than the floating point benchmarks.

totypes of low-latency coherence [40]. In fact, it is the minimum that is reasonably possible with a 4×4 2D mesh network. (Running microbenchmarks in my testbed, I found that Intel Ivy Bridge is 75 cycles, Intel Sandy Bridge is 95 cycles, and Intel Nehalem is 110 cycles.) I only use this low-latency model to simulate conventional hardware, and I later (Section 3.5.2) show that low latency alone is not enough to compensate for the lazy nature of the cache coherence protocol.

Simulated ring cache I extended XIOSim to simulate the ring cache described in Section 3.4. Unless otherwise noted, the simulated ring cache has the following configuration: a 1KB 8-way associative array size, a one-word data bandwidth, a five-signal bandwidth, a single-cycle adjacent core latency, and two cycles of core-to-ring-node injection latency to minimally impact the already delay-

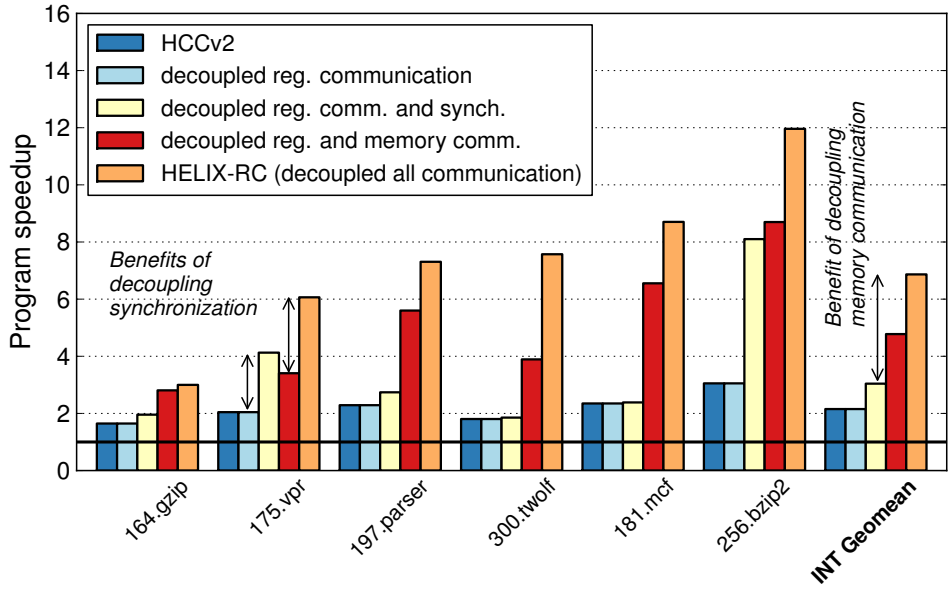



**Figure 3.10:** Decoupling register, synchronization, and memory communication is vital for maximizing speedups.

critical path from the core to the L1 cache. I used a simple bitmask as the hash function to distribute memory addresses to their owner nodes. To avoid triggering the cache coherence protocol, all words of a cache line have the same owner. Lastly, XIOSim simulates changes made to the core to route memory accesses either to the attached ring node or to the private L1.

Benchmarks I used 10 of the 15 C benchmarks from the SPEC CPU2000 suite: 4 floating point (SPECfp 2000) and 6 integer (SPECint 2000) benchmarks. For engineering reasons, the data dependence analysis that HCCv3 relies on [23] requires either too much memory or too much time to handle the other benchmarks. This limitation is orthogonal to the results described below.

Compiler I extended the ILDJIT compilation framework [8], version 1.1, to use LLVM 3.0 for backend machine code generation. I generated both single- and multi-threaded versions of the benchmarks. The single-threaded programs are the unmodified versions of the benchmarks, op-

timized ($O_3$) and generated by LLVM. This code outperforms GCC 4.8.1 by 8% on average and underperforms ICC 14.0.0 by 1.9%.[¶] The multi-threaded programs were generated by HCCv3 and HCCv2 to run on ring-cache-enhanced and conventional architectures, respectively. Both compilers produce code automatically and do not require any human intervention. During compilation, they use SPEC training inputs to select the loops to parallelize.

Measuring performance I computed the speedups relative to sequential simulation. Both single- and multi-threaded runs use reference inputs. To make simulation feasible, I simulated multiple phases of 100M instructions as identified by SimPoint [24].

### 3.5.2 Speedup Analysis

In the 16-core processor evaluation system, HELIX-RC boosts the performance of sequentially-designed programs (SPECint 2000), which are assumed not to be amenable to parallelization. Figure 3.9 shows that HELIX-RC raises the geometric mean of speedups for these benchmarks from 2.2× for HCCv2 without ring cache to 6.85×.

HELIX-RC not only maintains the performance increases of HCCv2 (compared to HCCv1) on numerical programs (SPECfp 2000), but also increases the geometric mean of speedups for SPECfp 2000 benchmarks from 11.4×[‖] to almost 12×.

Next, I turn to explaining where the speedups come from.

Communication Speedups obtained by HELIX-RC come from decoupling both synchronization and data communication from computation in loop iterations, which significantly reduces communication overhead, allows the compiler to split sequential segments into smaller blocks,

[¶] As an aside, automatic parallelization features of ICC led to a geomean slowdown of 2.6% across the SPECint 2000 benchmarks, suggesting ICC cannot parallelize irregular programs.

[‖] These speedups are possible even with the cache coherence latency of conventional processors (e.g., 75 cycles).

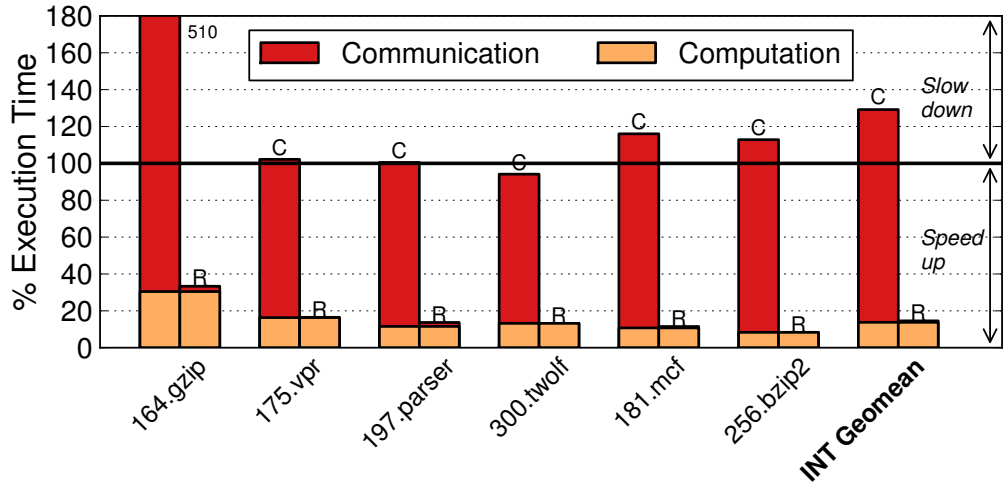



**Figure 3.11:** While code generated by HCCv3 speeds up with a ring cache (R), it slows down on conventional hardware (C).

and cuts down the critical path of the generated parallel code. Figure 3.10 compares the speedups gained by multiple combinations of decoupling synchronization, register-based communication, and memory-based communication. As expected, fast register transfers alone do not provide much speedup since most in-register dependences can be satisfied by recomputing the shared variables involved (Section 3.1). Instead, most of the speedups come from decoupling communication for both synchronization and memory-carried actual dependences. To the best of my knowledge, HELIX-RC is the only solution that accelerates all three types of transfers for actual dependences. The Multiscalar register file (Section 2.4.1) comes closest by decoupling register communication and synchronization, but as the figure shows, decoupling communication and synchronization for memory provides significant additional benefits.

In order to assess the impact of decoupling communication from computation in the SPECint 2000 benchmarks, I executed the parallel code generated by HCCv3—assuming a decoupling architecture like a ring cache—on a simulated conventional system that does not decouple. The loops selected under the assumption that a fast communication mechanism is present do require frequent

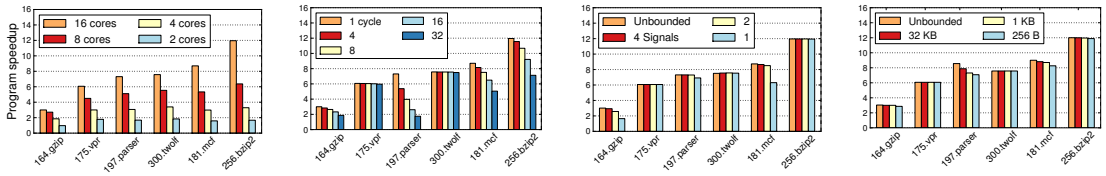
communication (every 24 instructions on average). Figure 3.11 shows that such code, when run on a conventional multicore processor (left bars), performs no better than sequential execution (100%), even with the optimistic 10-cycle core-to-core latency. These results further underscore the importance of selecting loops based on the core-to-core latency of the architecture.

Sequential segments While more splitting offers higher TLP (more sequential segments can run in parallel), more splitting also requires more synchronization at run time. Hence, the high synchronization cost for conventional multicore processors discourages aggressive splitting of sequential segments.[**] In contrast, the ring cache enables aggressive splitting to maximize TLP.

To analyze the relationship between splitting and TLP, I computed the number of instructions that execute concurrently for the following two scenarios: (i) conservative splitting constrained by a contemporary multicore processor with a high synchronization penalty (100 cycles) and (ii) aggressive splitting for HELIX-RC with low-latency communication (<10 cycles) provided by the ring cache. In order to compute the TLP independent of both the communication overhead and core pipeline advantages, I used a simple abstracted model of a multicore system that has no communication cost and is able to execute one instruction at a time. Using the same set of loops chosen by HELIX-RC and shown in Figure 3.9, TLP increased from 6.4 to 14.2 instructions per cycle with aggressive splitting. Moreover, the average number of instructions per sequential segment dropped from 8.5 to 3.2 instructions.

Coverage Despite all the loop-level speedups made possible by decoupling communication and aggressively splitting sequential segments, Amdahl's law states that program coverage dictates the overall speedup of a program. Prior parallelization techniques have avoided selecting loops with small bodies because communication would slow down execution on conventional processors [12, 56]. Since HELIX-RC does not suffer from this problem, the compiler can freely select

[**] This is the rationale behind DOACROSS parallelization [16].




**(a)** Core count.

**(b)** Adjacent node link latency.

**(c)** Signal bandwidth.

**(d)** Node memory size.

**Figure 3.12:** Sensitivity to core count and ring cache parameters. Only SPECint benchmarks are shown.

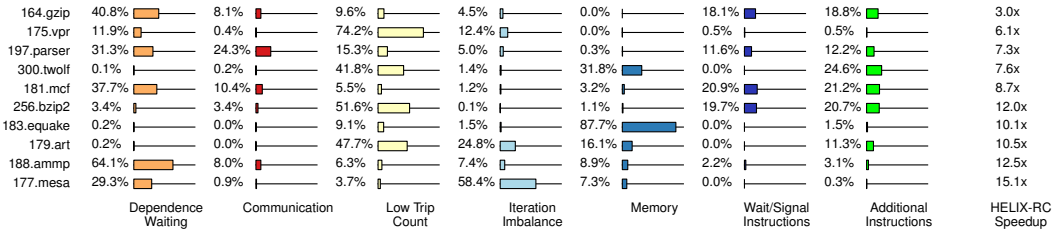



**Figure 3.13:** Breakdown of overheads that prevent HELIX-RC from achieving ideal speedup.

small hot loops to cover in almost the entirety of the original program. Table 3.1 shows that HELIX-RC achieves >98% coverage for all of the benchmarks evaluated.

### 3.5.3 Sensitivity to Architectural Parameters

The speedup results presented so far assume the default configuration (in Section 3.5.2) for the ring cache. I will now investigate the impact of different architectural parameters on speedup. In the next set of experiments I sweep one ring cache parameter at a time while keeping all others constant at the default configuration.

Core count Figure 3.12a shows that HELIX-RC efficiently scales parallel performance with the core count, from 2 to 16.

Link latency Figure 3.12b shows the speedups obtained versus the minimum communication latency between adjacent ring nodes. As expected, HELIX-RC performance degrades for longer

latencies for most of the benchmarks. It is important to note that current technologies can satisfy single-cycle adjacent core latencies, as confirmed by commercial designs [64] and CACTI [43] wire models of interconnect lengths for dimensions in modern multicore processors.

Link bandwidth A ring cache uses separate dedicated wires for data and signals to simplify the design. The simulations confirm that a minimum data bandwidth of one machine word (hence, a single write port) sufficiently sustains more than 99.9% of the performance obtained by a data link with unbounded bandwidth for all benchmarks. In contrast, reducing the signal bandwidth can degrade performance, as shown in Figure 3.12c, due to synchronization stalls. However, the physical overhead of adding additional signals (up to 4) is negligible.

Memory size Figure 3.12d shows the impact of memory size. The finite-size cases assume LRU replacement. Reducing the cache array size within the ring node only impacts 197.parser, which has the largest ring cache working set.

### 3.5.4 Analysis of Overhead

To identify areas for improvement, I have categorized every overhead cycle (preventing ideal speedup) based on a set of simulator statistics and the methodology presented by Burger and colleagues [7]. Figure 3.13 shows the results of this categorization for HELIX-RC, again implemented on a 16-core processor.

Most importantly, the small fraction of *communication* overheads suggests that HELIX-RC successfully eliminates the core-to-core latency for data transfer in most benchmarks. For several benchmarks, notably 175.vpr, 300.twolf, 256.bzip2, and 179.art, the major source of overhead is the low number of iterations per parallelized loop (*low trip count).* While many hot loops are frequently invoked, the low iteration count (ranging from 8 to 20) leads to idle cores. Other benchmarks, such as 164.gzip, 197.parser, 181.mcf, and 188.ammp, suffer from dependence waiting due to large sequential

segments. Finally, HCCv3 must sometimes add a large number of `wait` and `signal` instructions (i.e., many sequential segments) to increase TLP, as can be seen for 164.gzip, 197.parser, 181.mcf, and 256.bzip2.

## 3.6 Conclusion

Decoupling communication from computation makes irregular programs easier to parallelize automatically by compiling loop iterations as parallel threads. While numerical programs can often be parallelized by compilation alone, irregular programs greatly benefit from a combined compiler–architecture approach. The HELIX-RC prototype shows that a minimally invasive architecture extension co-designed with a parallelizing compiler can liberate enough parallelism to make good use of 16 cores for irregular benchmarks commonly thought not to be parallelizable. Unlike the previous approaches described in Chapters 1 and 2, HELIX-RC decouples all inter-thread memory, register, and synchronization communication from the core computation.

In the next chapter, I delve into some of the important implementation details for ring cache.

# 4

# Ring Cache Detail and Implementation

In Chapter 3, most of the HELIX-RC speedup results were generated using a C/C++ based x86 simulator called XIOSIM [32], with the ring cache similarly modeled in C++. While every effort was made to accurately model the ring cache at a cycle-accurate level, high-level languages are not the best fit for expressing the operation of hardware. In order to fully specify the implementation of the ring cache, I created and tested a synthesizable Verilog reference design. Fully detailed explanations of the implementation and the design decisions appear in the Appendix to this dissertation.

In this chapter, I delve into some of the implementation details. First, I discuss the ring cache's integration with the normal cache hierarchy in more detail, along with a very important memory consistency issue that must be handled properly. Then I describe the signal buffer, which plays a crucial role in reducing HELIX-RC synchronization costs. I introduce the concept of *synchro-*

*nization epochs* and describe how the signal buffer can be designed to allow cores executing parallel code to decouple their execution by an arbitrary number of iterations. Finally, I describe the power and area consumed by the synthesized ring cache design and specifically examine some of the major tradeoffs related to the signal buffer.

## 4.1 Memory Hierarchy Integration

One of the most important contributions of ring cache is that it transforms a *reactive* cache coherence protocol into a *proactive* system of communication. In most cases, when a core attempts to access shared data, it will find that the data is present in its local ring node memory, and the communication cost is simply the time it takes to access the ring node. In contrast, if the normal cache hierarchy were used, it would take dozens of cycles from the time the data was requested until it arrived locally.

A distinguishing factor of ring cache relative to other fast communication mechanisms (Multiscalar register file [54], scalar operand networks [59]) is that the number of shared pieces of data is not known at compile time, nor is the number of consumers of any particular shared piece of data. Since other solutions for fast communication of dependences rely on statically known numbers of shared elements, they are not suitable for HELIX. Instead, a cache structure that can handle unknown numbers of elements is needed. This means that ring cache must be able to handle load misses and cache evictions, since there is no guarantee that all of a shared piece of data will fit in the ring cache. For this reason, the normal cache hierarchy must be relied upon to support the ring cache.

However, the memory consistency guarantees of the normal cache hierarchy are different than that of ring cache, and a naive integration between the two could raise a significant consistency issue. Consider an implementation where a ring node simply writes any evicted value back to its local L1. In the case of a subsequent ring cache load miss, the ring node fetches the data from its L1. While

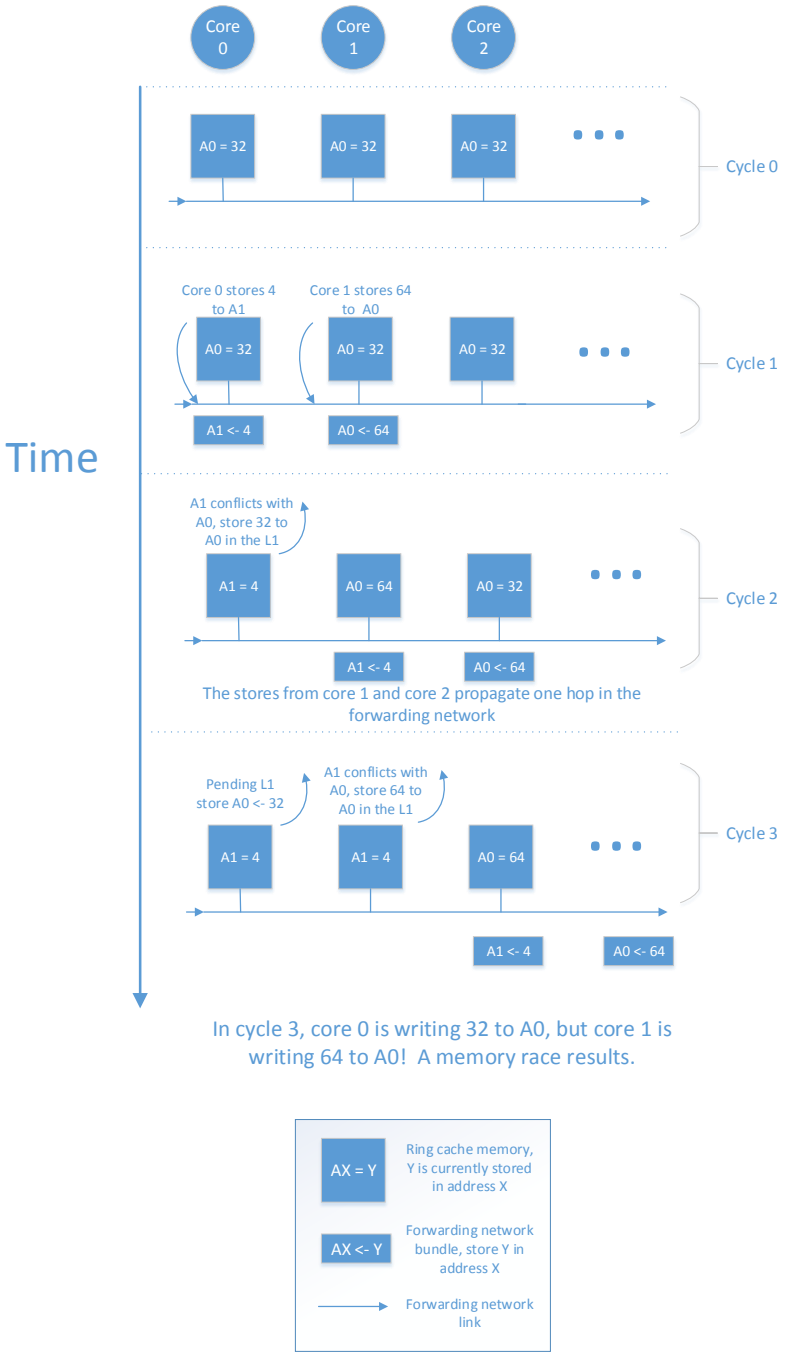



**Figure 4.1:** Allowing any core to load from and write back to the normal cache hierarchy results in a race condition that may violate correctness!

this might seem like a reasonable idea at first glance, it gives rise to race conditions that can violate correctness. Figure 4.1 depicts the timeline of a three-core system suffering from such a race condition. For simplicity, assume a ring node memory size of just a single word. Sometime in the past, the value 32 was written to address A0, which was propagated on the data/signal forwarding network and stored in every ring node memory. During cycle 1, core 0 and core 1 execute two different sequential segments. Core 0 stores the value 4 to address A1, and core 1 stores the value 64 to address A0. Both stores enter the data/signal forwarding network. In the next cycle, core 0's store has triggered an eviction in its memory, and A0, with a value of 32, begins to be written back to core 0's L1 cache. In the same cycle, core 1's ring node memory updates address A0 with the value 64. Additionally, both stores propagate one more hop on the forwarding network. In cycle 3, the store to A1 triggers an eviction of A0 in core 1. Now core 1 begins to write A0 back to its own L1. Unlike core 0, however, core 1 writes back the newly updated value of 64. This results in a race to update A0 with either the old or the new value. Many (perhaps all) modern architectures do not make any guarantees about which value will be recorded first.

I handle this consistency issue by enforcing the constraint that for any unique memory address, there is a single *owner core* that is solely responsible for any loads or stores to that address between the ring cache and the normal cache hierarchy. The owner core is determined based on certain bits of the address, depending on how many cores there are in the system.

### 4.1.1 Request and Reply Networks

In the case of a ring cache load miss, the request network is responsible for requesting the data from the owner ring node, which subsequently performs an L1 lookup and returns the result on the reply network. In the case of a ring cache eviction, if the ring node owns the evicted address, it writes it back to its L1. If the evicting ring node is not the owner, it simply discards the data without performing any write back. These networks implement a reactive data transfer mechanism that, if used

frequently, completely eliminates all of the benefits of having the ring cache. For this reason, they are not tuned for performance—if they are used more than rarely, performance will tank regardless. Like the data/signal forwarding network, the request/reply networks are also implemented with unidirectional ring networks that single hop around each core in the system, one clock cycle per hop. While it might seem that a more highly connected topology could have been used, since strict in-order data flow is not required for loads as it is for stores and signals, there are two reasons why I stayed with unidirectional rings. First, unidirectional rings are easy to reason about in terms of data flow and deadlock avoidance. Second, and more importantly, care must be taken so that a remote load on the request network doesn't pass a store to the same address on the forwarding network, or an incorrect value could be returned.

### 4.1.2 Reducing Remote Loads

The number of remote loads can be optimized by relaxing the constraint that only the *owner core* of an address can interact with the normal cache hierarchy. Instead, only the owner core of an address can interact with the normal cache hierarchy *if the address has previously been written to ring cache but subsequently evicted*. The race condition previously shown in Figure 4.1 can only occur if an address was at some point present in the ring cache—in that example, address A0. If a particular core is trying to load A0 and knows for sure that A0 was *never* written to the ring cache before this loop invocation, it can conclude that it is impossible that it is currently being evicted from any other node. Given the definition of a sequential segment, if a core is loading A0, no other core in the system can be loading or storing it. It is therefore safe for the core to load it from the normal cache hierarchy, even if it isn't the owner. We use a bloom filter in each ring node to track whether an address has been written before this loop invocation. If a core attempts to perform a load, and it misses it in its ring node memory, the bloom filter is consulted. If the address is present in the bloom filter, the core knows it must make a remote load request unless it is the owner of the address. If the address is

not present in the bloom filter, the core can load from its own L1, despite not being the owner.

## 4.2 Signal Buffer Implementation

The signal buffer contributes a significant portion of the improved speedups that ring cache provides for HELIX. The signal buffer produces speedups both by pushing the signal tracking logic to hardware instead of software and by decoupling signal forwarding from synchronization, which helps break the sequential forwarding chains of synchronization that are intrinsic to HELIX and DOACROSS-style parallelization. The amount of hardware resources dedicated to the signal buffer can increase speedups along two dimensions. First, adding more available signal IDs allows the compiler to more aggressively parallelize sequential code into smaller sequential segments, potentially increasing parallelism amongst segments. Second, adding more bits for buffering received and sent signals allows cores to increase the number of iteration *epochs* they can decouple from one another during execution, which reduces the core idle time that normally results from sequential forwarding chains. I begin the signal buffer discussion by explaining the concept of epochs in this context, as well as how the signal buffer facilitates synchronization decoupling by allowing cores to skip *light waits*. Then I discuss some optimizations the compiler can make to reduce the amount of synchronization instructions that need to be sent to the signal buffer.

### 4.2.1 Synchronization Epochs

Section 3.1.2 described at a high level how decoupling signal forwarding from synchronization allows HELIX-RC to break sequential forwarding chains and increase speedups. In this section, I describe in detail the operation of the signal buffer, how it breaks these chains, and how it allows cores to decouple in units of epochs.

I define an *epoch* to be a set of N iterations, where N is the number of cores executing a parallel loop. Consider a 3-core system. For a parallel loop with 8 total iterations, core 0 executes iterations

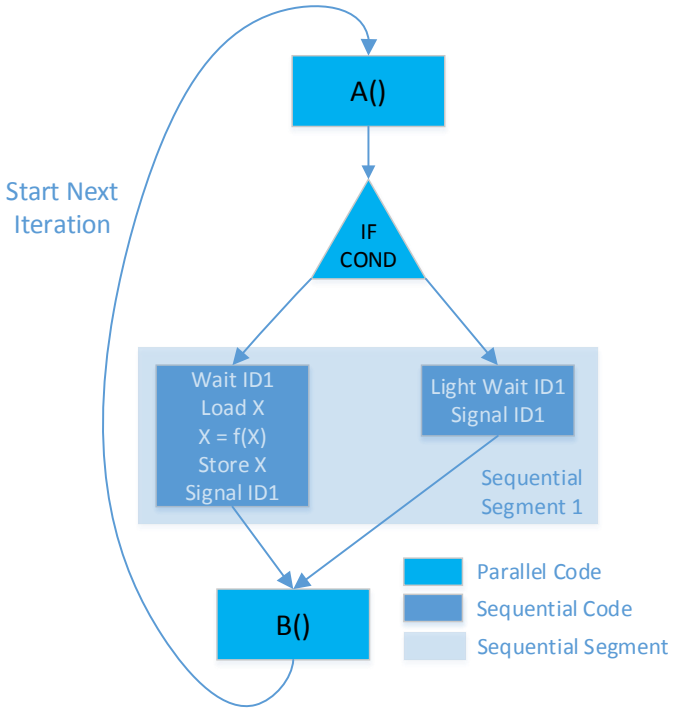



**Figure 4.2:** A modified loop body where empty sequential segments start with *light wait* instructions instead of ordinary wait instructions.

0, 3, and 6; core 1 executes iterations 1, 4, and 7; and core 2 executes iterations 2, 5, and 8. An epoch is therefore 3 iterations long. The start of an epoch, however, does not need to occur at the beginning of a sequence of iterations. For example, an epoch could span from iteration 1 to iteration 3, not just from 0 to 2.

I further define the concept of a *synchronization epoch*, which is an epoch whose bounds are not at iteration boundaries. Instead, they exist just before the next sequential segment in an iteration. Take the example of an epoch of iterations from 1 to 3. The corresponding synchronization epoch would begin precisely at the next sequential segment encountered by the core executing iteration 1, and would end just before the next sequential segment encountered by the core executing iteration 3. In higher-level terms, a synchronization epoch is the furthest distance any pair of cores can drift apart if they are constrained by a sequential forwarding chain. In the case where all sequential segments contain dependences that need to be satisfied, no two cores can ever separate by more than a synchronization epoch, even if signal buffering is available, since all sequential segments must be executed in loop iteration order. However, if there are sequential segments that are empty and there is sufficient signal buffering available, cores can drift apart ("decouple") by multiple synchronization epochs, which reduces core idle time. I define wait instructions that mark the boundaries of such empty sequential segments as *light waits* that do not need to be synchronized under certain circumstances.

An example will be helpful for understanding these two concepts. Figure 4.2 depicts a HELIX parallelized loop, a modified version of Figure 3.7(a), with a `light wait` instead of a normal `wait` instruction on the right branch of the sequential segment, indicating that the segment lacks any dependence. To keep things simple, assume we have a two-core system—the same conclusions will hold for a chip with more cores. Each core only needs to receive signals from the other core to unblock a particular sequential segment. First, we consider a scenario where a core uses a single bit per core for each other core in the system to track signals received from that core for a particular sequen-

tial segment. Since we have only two cores and only one sequential segment, this means that each core only needs a single bit to track signals. When a core receives a signal, it sets the signal bit. When a core finishes executing the sequential segment, it clears the bit, thereby "consuming" the signal.

Figure 4.3 depicts the execution of four iterations of the example loop. The time it takes to transmit a signal is exaggerated to better illustrate the impact of signal buffering. Note that since core 0 is executing the first iteration of a loop, its signal bit is preset, since there is no iteration -1 to receive a signal from. For simplicity, data communication is not shown. First, core 0 and core 1 start executing the parallel portions of iterations 0 and 1. Just before reaching the sequential segment, core 0 is stalled on a DRAM access. Meanwhile, core 1 reaches the `light wait` instruction at the start of the sequential segment. Core 1 executing iteration 1 hasn't received the signal from core 0 executing iteration 0, so it may not enter the sequential segment. However, since it took the right branch of the *if*, the sequential segment starts with a `light wait` rather than a normal `wait`. Core 1 therefore knows that it doesn't contain a dependence to synchronize and would prefer to continue with execution, even though it is blocked. In this situation, core 0 and core 1 are said to both be within the same synchronization epoch. Once the DRAM access returns, core 0 executes the segment, clears its signal received bit, and sends the signal to unblock core 1. Core 1 sets the signal received bit, which grants it access to the sequential segment, before quickly sending its own signal, which clears the received bit. Core 0 soon begins executing iteration 2, and core 1 begins executing iteration 3, where the same dynamic repeats itself, although without the DRAM stall. Even though core 1 never needs to access shared data, it is nonetheless constrained by the sequential segment. When HELIX does not have access to a ring cache, cores will always be constrained to a single synchronization epoch. This can be seen by observing that throughout, iteration 0 only overlaps with iteration 1, which in turn only overlaps with iteration 0 and iteration 2.

Imagine a scenario where core 1, knowing that the sequential segment doesn't contain a dependence, bypasses the `light wait` instruction altogether. It would send the corresponding signal,

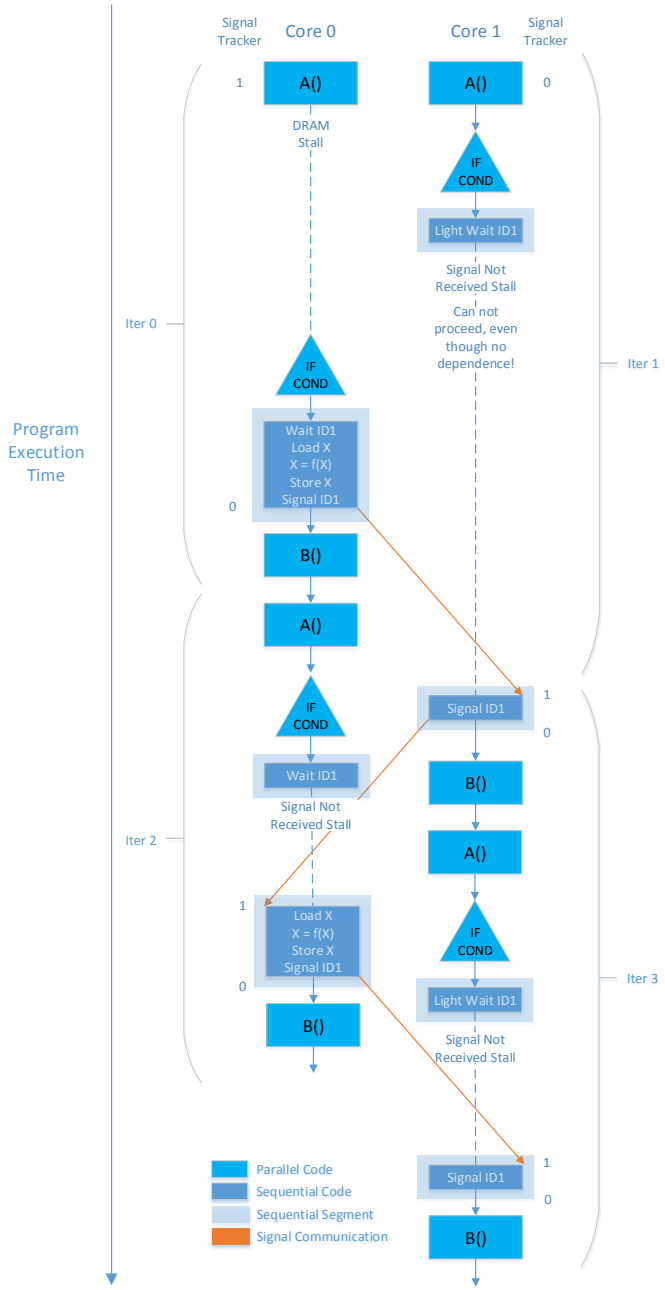



**Figure 4.3:** A single state bit for synchronizing signals constrains cores to operate within a single synchronization epoch. In a two-core system, this implies that cores cannot move apart by more than two iterations.

which would ordinarily clear core 1's bit, and set core 0's bit. However, core 1's bit is already cleared and core 0's bit is already set. In this case, the information that a signal was consumed and that a signal was received is lost. Core 1, upon receiving the signal from core 0 (executing iteration 0), will set its bit and therefore enter the sequential segment of iteration 3, even though that signal was meant for the segment it skipped in iteration 1. This potentially results in accessing shared data out of iteration order, a correctness violation. Likewise, core 0 has lost the knowledge that it received a signal from core 1, iteration 1, and therefore may not enter the sequential segment in iteration 2. Since we would like core 1 to be able to skip the empty sequential segment, we add an additional bit to our signal tracking. Instead of 2 states (received signal or not), there are now 4. These new states allow cores to record whether they've skipped a sequential segment (state -1) and therefore need to receive two more signals to enter the next non-light sequential segment, and also whether they've received an extra signal (state 2) and therefore are free to enter the sequential segment two more times.

Figure 4.4 depicts a new execution timeline for when this additional signal-buffering hardware is added. This time, core 1 is able to skip the sequential segment and race ahead to iteration 3 without violating correctness. Core 1 takes the right branch of the *if* and, even though the sequential segment is once again empty, must block. However, because of the extra signal-buffering capability, the cores have now been able to drift apart by an additional synchronization epoch, allowing core 1 to begin executing iteration 3 before iteration 0 has executed the sequential segment. For every two additional states that are added to track signals, cores can drift apart yet another synchronization epoch. Of course, cores can only decouple if they encounter sequential segments that are empty—otherwise, they need to access shared data and the sequential segments still need to be executed in loop iteration order. The benchmarks contain enough empty segments, however, that decoupling cores reduces idle time and increases speedups—significantly so for some benchmarks, as we saw in Figure 3.10. In our simplified example, the execution time only improves slightly. In more realistic scenarios, with more cores and more heterogeneity in execution, having cores execute as far ahead

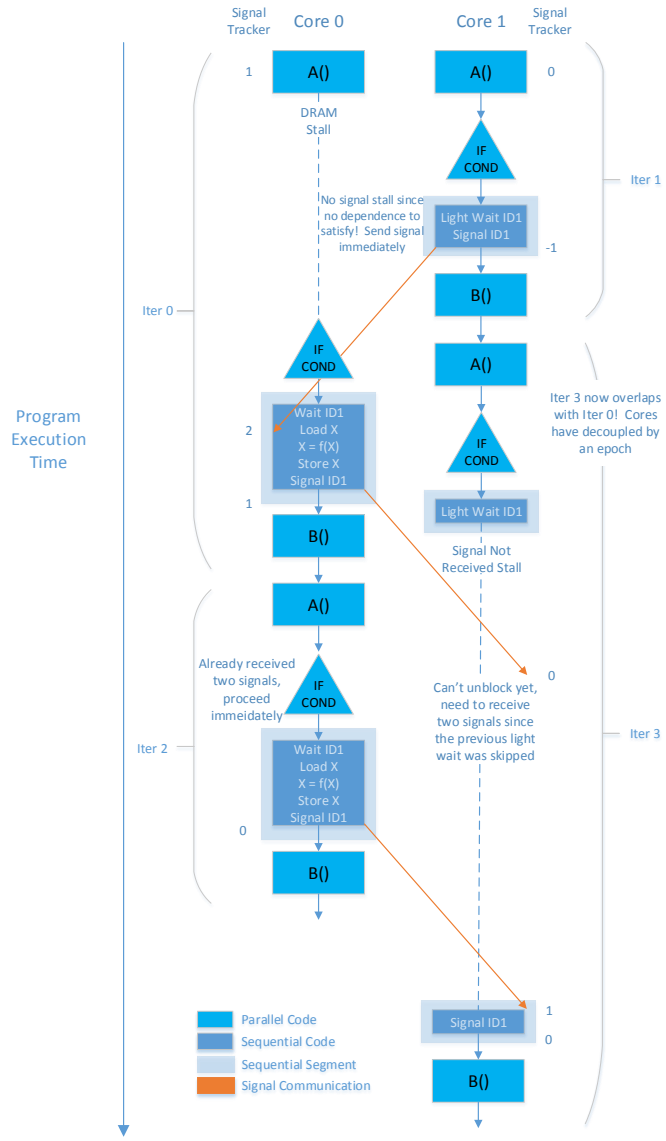



**Figure 4.4:** Using two signal-buffering state bits improves performance by allowing cores to decouple by an additional synchronization epoch.

as possible and send as many signals as soon as possible increases overall performance even more. In practice, even given unconstrained resources, the maximum that cores drift apart in the complex benchmarks is usually limited to two synchronization epochs, so only four states for signal buffering are required to gain most of the benefit. I call the number of synchronization epochs by which the signal buffer can decouple the *epoch bound*.

### 4.2.2 Signal Buffer Architecture

Although the previous example had only two cores, the signal buffer must independently track signals from every other core in the system, whatever the number, not just a single core. Figure 4.5 shows the general hierarchy of the signal buffer. Each signal buffer within a single ring node contains signal tracker modules equal in number to the maximum number of signal IDs that the compiler can create. Within each signal tracker module, there are core tracker modules, one for each core in the system other than the core that the signal buffer is part of. Each of these core tracker modules contains a counter large enough to represent the total number of states required for the desired amount of synchronization decoupling, with two states per desired epoch of decoupling. As mentioned earlier, if the epoch bound is 2, four states are required to buffer signals and so a two-bit counter is used.

### 4.2.3 Signal Buffer Optimization

There are a few possible signal buffer optimizations that may be appropriate to use with the reference design. The first of these is only applicable when the epoch bound is 1—that is, cores are not able to decouple and must always stay within the same synchronization epoch, even in the presence of sequential segments that lack dependences. This scenario is similar to when HELIX is running on a traditional multicore, albeit still with faster signal propagation speed. In this configuration, every sequential segment is run in loop iteration order, without exception. `Light waits` don't ex-

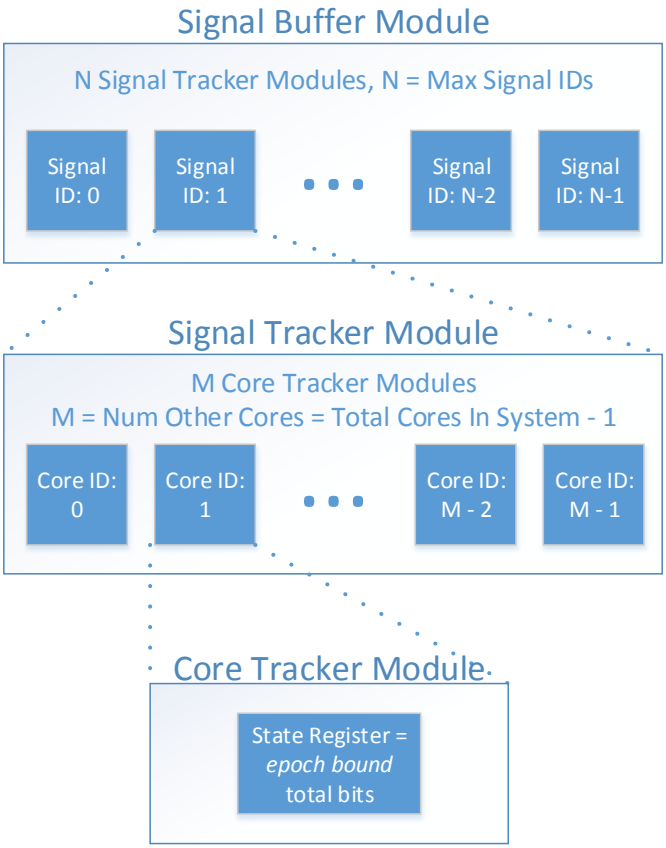



**Figure 4.5:** The signal buffer of a single ring node contains signal tracker modules for the total number of possible signal IDs that the compiler might generate. Within each signal tracker, there are core tracker modules, one for each *other* core in the system that the owner of this signal buffer might receive signals from. Within these modules, state registers track how many signals have been received relative to how many have been sent by the owner core. The number of necessary states is dictated by the desired amount of synchronization decoupling.

ist, since there is no situation where they can be skipped. The requirement for (numCores - 1) core tracker modules per signal tracker module is no longer necessary, since receiving a signal for a particular sequential segment ID from a particular core implies that every previous loop iteration from every other core has already executed the segment. This allows the signal buffer to reduce the number of core tracker modules from (numCores - 1) per signal tracker module to just 1 per signal tracker module. This potentially reduces the total amount of signal buffer area by a factor of the number of cores. The area savings come at a cost, however: cores cannot decouple across synchronization epochs and so speedups are reduced. However, if the signal buffer area is prohibitively large, limiting the epoch bound to 1 is a possible solution. Also, signals no longer need to circulate around the entire ring; they only need to travel to the subsequent core in the ring, which reduces signal bandwidth requirements.

The second optimization is applicable only when the epoch bound is 2, which implies cores can decouple an additional synchronization epoch. If the compiler can guarantee that there is at least one non-light `wait` instruction per loop iteration (as would be the case if a particular dependence always needed to be satisfied), then `light waits` can be removed entirely, leaving only the corresponding `signal` behind. This non-light `wait` instruction *must* belong to the same sequential segment in every iteration. A core can just straight away send the corresponding `signal` without relying on the `light wait` to prevent underflow. This is because the existence of at least one unavoidable non-light `wait` per iteration naturally limits synchronization decoupling to less than two synchronization epochs. This optimization is a trade-off—`light wait` instructions can be removed, which reduces the number of instructions executed, and the signal buffer no longer needs to contain the logic to examine `light wait`s. However, the compiler may now have to artificially generate a non-light `wait` instruction that could otherwise could be light. In practice, I found that this downside doesn't happen often. This optimization was used in the HELIX-RC evaluation in Chapter 3.

The third optimization applies when the epoch bound is 3. As with the previous optimization, if the compiler can make a guarantee about non-light `wait`s, than `light wait`s don't need to be executed at all. However, in this case, the compiler only needs to guarantee that at least one sequential segment executes a non-light `wait` per iteration. Unlike the previous case, it needn't be the same sequential segment in every iteration. This prevents our six-state core tracker counters from ever underflowing. For our particular benchmarks, increasing the epoch bound beyond 2 did not have any effect on performance, so this optimization may not be useful.

## 4.3 Ring Cache Synthesis Evaluation

**Table 4.1:** Ring cache parameters for the reference design.

| | |
|---|---|
| Number of Supported Cores | 16 |
| Address Width | 32 bits |
| Data Width | 32 bits |
| Cache Associativity | direct mapped |
| Cache Data Storage | 1024 KB |
| Total Number of Signal IDs | 128 |
| Signal Bandwidth | 5 signals per cycle |
| Signal Buffer *epoch bound* | 2 |
| Store Bandwidth | 1 per cycle |
| Data/Signal Forwarding Network Total Wires | 129 bits |
| Remote Load Request Network Total Wires | 37 bits |
| Remote Load Reply Network Total Wires | 37 bits |
| Assumed Network Link Latency | 0.5 ns |

This section presents some preliminary area, power, and timing results for the ring cache. Table 4.1 shows the parameters for the reference design. The values were chosen to roughly match the simulated ring cache in Chapter 3. The most noticeable exception is the ring cache memory, which was 8-way set associative in Chapter 3 rather than direct mapped. Later simulations showed that the difference between 8-way associativity and direct mapping was minor, so for ease and clarity of

implementation, I used the latter. It is important to note that many of these parameters were selected to be just large enough so as not to be a bottleneck for the six SPECint 2000 benchmarks that were evaluated. Other programs may have vastly differently requirements, so these particular values should not be overly relied upon for a final implementation.

The reference configuration was exhaustively tested with test vectors from our team's X86 cycle-level C++ simulator, XIOSIM, with the modeled ring cache configured the same as the reference design. Vectors were collected for every Simpoint phase of all 10 of the SPEC benchmarks that were evaluated in Chapter 3. Specifically, for every simulated cycle, the values of all inputs and outputs corresponding to the ring cache interface between the attached core and the L1 cache were collected. Verilog simulations were performed with 16 ring nodes that were excited with these test vectors. At every cycle, the outputs of the ring nodes were compared to the known correct outputs from XIOSIM. Every phase passed the testbench.

Many of the parameters, such as number of supported cores, signal bandwidth, signal buffer configuration, and total number of signal IDs, have a large impact on the area/performance of the ring cache. After generating initial results for this reference design, some of these important parameters were swept. First, the reference design was synthesized using Synopsys Design Compiler with a 40nm process technology. The synthesis tool was steered to optimize for critical path delay. I used RTL-level activity factors from the SPECint test runs to more accurately predict power in the synthesized design. Table 4.2 summarizes the post-synthesis results for a single ring node. Although a 0.5 ns link latency was assumed for all of the inter-node links, I stress that the power and area numbers here do not account for any link area/power and strictly represent only Design Compiler's post-synthesis estimates. By using RTL simulation activity factors instead of Design Compiler's default, the power is reduced from nearly 100 mW to 19.22 mW—because although the SPECint benchmarks use the ring cache relatively regularly, it is still only accessed in sequential segments. Since there are still significant portions of parallel code, the ring cache is often dormant.

**Table 4.2:** Synthesis results for a single reference ring node.

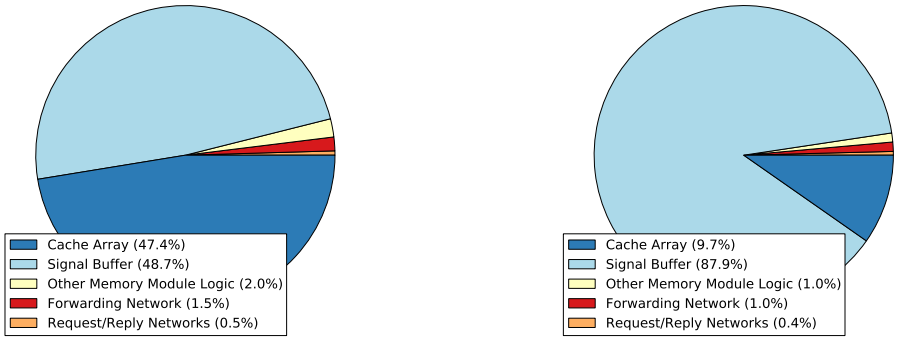
| | |
|---|---|
| Area | 0.272 sq mm |
| Dynamic Power | 19.22 mW |
| Leakage Power | 3.3 mW |
| Max Frequency | 1.11 GHz |

### Critical Path

The post-synthesis timing report exposes two primary critical paths in the ring cache design. The first path starts in the network receive buffers for the primary data/signal network, for a bundle of stores/signals that may be sent to the subsequent ring node during this cycle. From there, the path continues through the logic that decides 1) whether a new store/signal from the core can be added to the aforementioned network bundle and 2) whether any circulating stores/signals have completed a full cycle and must therefore be removed. Finally, the critical path extends over the outgoing network link to the next core. Since link propagation accounts for 0.5 ns of the path, the routing logic accounts for only a small portion of the total path. It is beneficial that link propagation can happen in parallel with writing to the memory, since the next longest paths involve the memory. Specifically, they start at the data/signal network receive buffers as before. But then the paths change and instead go to the memory array to perform a tag lookup and prepare a cache array write for the next clock edge.

### Area

Figure 4.6a depicts the area usage in the reference design. The cache array is marginally smaller than the signal buffer. Although perhaps unexpected, this follows from the fact that the reference design uses a total of 128 signal IDs. Since the storage per ID is 2 storage bits per core per signal, the total number of registers per signal buffer is 2 * 16 * 128 = 4096 bits. The area of the memory is somewhat larger than it could otherwise be, since the reference design uses a register array in lieu of an SRAM




**(a)** Ring Node Area

**(b)** Ring Node Dynamic Power

**Figure 4.6:** Power and area for a single ring node. The forwarding network includes all logic to route data/signals between nodes. The request/reply networks include all of the logic necessary to integrate the ring cache with the rest of the normal memory hierarchy.

to minimize access latency. If the area is prohibitively large, it could be very beneficial to experiment with using an SRAM (which would necessitate adjusting the FSMs in the memory and array modules) and reducing the number of signal IDs.

#### Power

Figure 4.6b shows the dynamic power breakdown. Although the cache array constitutes a large portion of the area, it constitutes a relatively smaller proportion of the power consumption. The signal bandwidth (5 signals per cycle) was set at this high level relative to the store bandwidth (1 store per cycle) because the large number of empty sequential segments tends to produce far more signals than shared data. As a result, the signal buffer is utilized far more frequently than the cache array.

### 4.3.1 Signal Buffer Parameter Sweeps

The signal buffer has several parameters that potentially have a large impact on system performance and ring node area. These include the total number of possible signal IDs, the amount of signal bandwidth, the amount of allowed synchronization epoch decoupling, and the number of cores in the system.

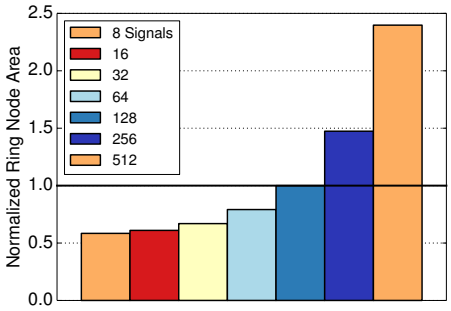



**Figure 4.7:** Total ring node area as total signal ID capacity is swept from 8 to 512.

### Number Of Signal IDs

The compiler has control over the maximum number of sequential segments it will produce in any given loop. Maximum performance can be achieved when the compiler has total flexibility to create as many sequential segments as it would like. If restricted, the compiler must combine multiple sequential segments into one, which may have an impact on performance. However, the number of signal IDs has a linear effect on the amount of signal buffer area, as each ring node must contain signal tracker modules for each possible unique signal ID. In Chapter 3, the number of signal IDs was unrestricted, and a maximum of approximately 128 signal IDs was required for most loops. Due to current limitations in the compiler, the maximum number of signal IDs cannot be swept at this time. I leave this analysis for future work. However, although I can't examine this thoroughly, the intuition I have gathered from examining the compiled code suggests that significantly fewer than 128 signals would be required to capture most or all of the performance. Some of the sequential segments are purely for edge cases (exceptions, error handling) that do not occur in normal operation and are rarely if ever synchronized.

However, I was able to sweep the maximum number of signal IDs in the signal buffer to see how the area changes. Figure 4.7 depicts the area of a ring node, normalized to the reference design, as the number of signals are swept from 8 to 512. Given that the signal buffer was a significant fraction of

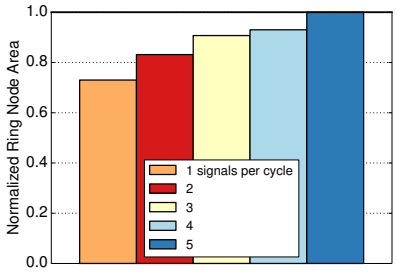



**Figure 4.8:** Increasing signal bandwidth increases signal buffer and network buffer sizes.

the total ring node area in the reference design, it is no surprise that the total ring node area increases dramatically for the largest signal capacities.

### Signal Bandwidth

Figure 3.12c from Chapter 3 shows the importance of high signal bandwidth for achieving good speedups on SPECint. Although it doesn't have as drastic an effect on area as the number of signal IDs, increasing the signal bandwidth does increase the area of the signal buffer. Additional multiplexers and logic to process multiple incoming signals result in a 28% decrease in total ring node area between the reference design and one with a signal bandwidth of 1 per cycle, as seen in Figure 4.8. Since the signal buffer is approximately 50% of the design area, this corresponds to about a 55% decrease in signal buffer area. Although the data/signal network buffers are also halved, their overall impact is very slight, given their size. Since the critical path involves these network buffers, the maximum achievable frequency increases slightly, by around 5%, as signal bandwidth decreases from 5 signals per cycle to 1 signal per cycle.

### Amount of Synchronization Decoupling

The epoch bound parameter in the signal buffer dictates how many synchronization epochs can be decoupled by cores. In Chapter 3, this parameter was essentially set to 2, with the optimization de-

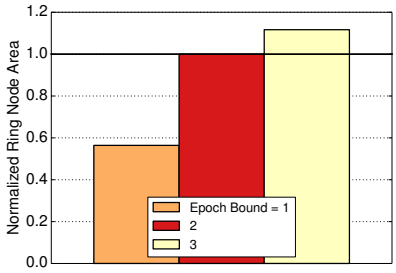



**Figure 4.9:** Decoupling synchronization from one to two epochs increases area significantly, but also increases performance.

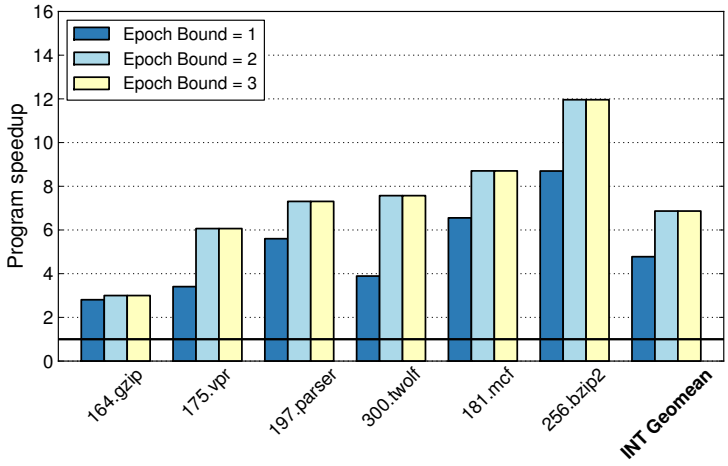



**Figure 4.10:** Decoupling synchronization up to two epochs increases speedups, but decoupling any further has no effect.

scribed in Section 4.2. Increasing this parameter has the potential to increase speedup, but will also increase the area consumed by the signal buffer, as more bits are needed to track the state of each signal. Figure 4.9 shows the area impact for three values of the epoch bound: 1, 2, and 3. Note the large impact of moving from 1 to 2. This happens because at an epoch bound value of 1, the signal buffer can be simplified by having each core only track received signals from its immediate predecessor and only send signals to its immediate successor. This optimization is possible because at an epoch bound value of 1, cores are unable to decouple. This implies executing all sequential segments

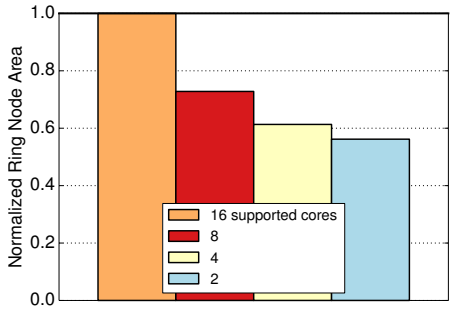



**Figure 4.11:** Signal buffer size varies linearly with the number of supported cores, so decreasing the number of cores has a large impact on the ring node area.

strictly in loop iteration order, which removes the reason for having a core broadcast a signal to every other core, as receiving a signal from your immediate predecessor guarantees that all previous iterations have already executed older iterations. However, an epoch bound of 2 has a drastic performance impact, as seen in the simulation results in Figure 4.10 (this consists of some of the same data as Figure 3.10). In contrast, moving to an epoch bound of 3 has absolutely no benefit—there are only a very few times in all of the combined SimPoint phases where any benchmarks decouple by that amount. Unless other program characteristics vary significantly from SPECint, it seems pointless to use any epoch bound value other than 2.

### Number of Supported Cores

The signal buffer needs to track received signals from every other core in the design. Consequently, the amount of state in the signal buffer varies linearly with the number of cores in the system, in much the same way that the total number of signal IDs does. Figure 4.11 shows the drastic decrease in ring node size when the number of cores is reduced from 16 to 2. The size of the signal buffer decreases by a factor of 8×. Intuitively, as the number of cores decreases, so does the achievable speedup, as I previously showed in Figure 3.12a.

## 4.4 Conclusion

A number of important engineering decisions must be made when designing the ring cache, many of them highly dependent on the characteristics of the workloads being parallelized. Appendix A contains the full details, schematics, control FSMs, and more for all of my ring cache design decisions.

# 5

# Future Directions for HELIX-RC

With the primary communication bottleneck of HELIX solved with ring cache, there are a variety of possibilities for future studies. In this chapter, I present some initial results from two such studies, in addition to proposing additional work that could be performed with some modification to the compiler. First, since the original evaluation of HELIX-RC involved only in-order cores, I examine how much performance is lost when switching to out-of-order cores. I conclude that although the out-of-order cores reduce the amount of TLP that HELIX-RC can extract, performance does not suffer greatly on more complex cores. Second, I compare traditional multiprogramming parallelism with HELIX-RC automatic single program parallelism. Intuitively, one would suspect that HELIX-RC parallelized programs would make worse use of cores than multiple independent program copies, since HELIX-RC speedups don't scale linearly with the number of cores. However, I

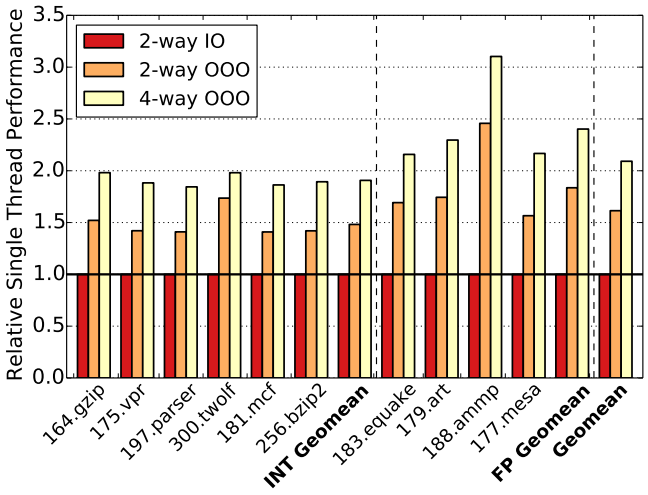



**Figure 5.1:** Singled-threaded SPECint 2000 performance increases by around 2× when moving from in-order to out-of-order cores.

show that, counterintuitively, HELIX-RC can make better use of multiple cores than merely running multiple copies of a program, since the burden on shared chip resources is reduced.

## 5.1 HELIX-RC With Out-of-Order Cores

HELIX-RC successfully extracts TLP for in-order cores. Out-of-order cores also automatically extract parallelism, in the form of finer-grained instruction-level parallelism (ILP). It is important to evaluate whether the ILP provided by modern complex cores eats away at TLP, reducing the speedups obtained by HELIX-RC. I compared the speedups on a 2-way and a 4-way out-of-order core to the speedup of the 2-way in-order Atom core. Although speedup relative to the single-threaded execution on a particular core type is somewhat reduced (from 8.5× on the in-order core to 7× on the 4-way out-of-order core), the overall performance of the parallelized code always improves as the core becomes more capable.

In this section I present the characteristics of the single- and multi-threaded SPEC 2000 benchmarks on different core types. I describe why out-of-order speedups decrease, and I detail possible

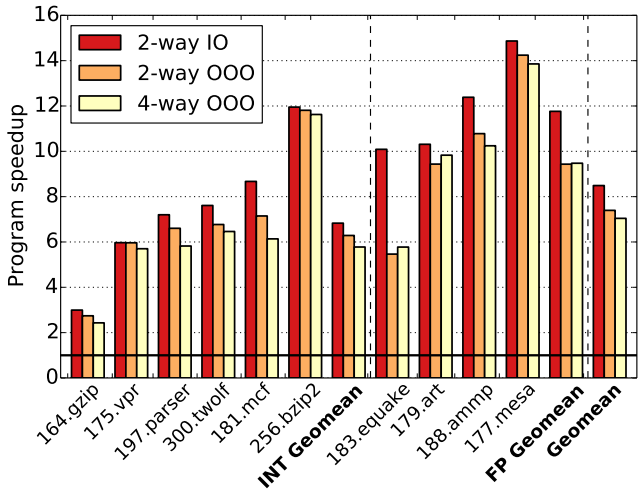



**Figure 5.2:** HELIX-RC speedup decreases from 8.5× to 7× on more complex cores, compared to their single-threaded executions.

solutions for restoring performance.

### 5.1.1 Out-of-Order Execution

For the purpose of comparison, I consider three different core types: a 2-way in-order Intel Atom, a 2-way out-of-order core, and a 4-way out-of-order core. The architectures of the out-of-order cores roughly correspond to a modern Intel Atom and an Intel Nehalem core, respectively. Since we are primarily concerned with the consequences of changing architecture, the core frequencies remain fixed for all three core types. The same 8 MB last-level cache from the default 16-core configuration is used, as is the 4 memory controller DDR3 memory configuration.

Figure 5.1 shows the relative single-threaded performance of the SPECint 2000 benchmarks I used for the three core types. The performance increase from the 2-way in-order to the 2-way out-of-order core is approximately 60%. Increasing the out-of-order commit width from 2 to 4 boosts the performance by an additional 45%. The floating point benchmarks generally see more of a performance improvement, as their memory accesses and control flow are more regular and predictable.

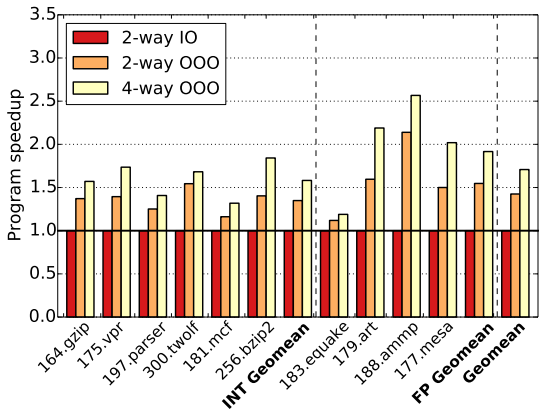



**Figure 5.3:** Absolute parallelized program performance always increases for HELIX-RC when moving from a simple 2-way in-order core to a 4-way out-of-order core—as much as 2.6× for 188.ammp and as little as 1.2× for 183.equake.

Although the out-of-order cores can extract more ILP for the same workloads, they also have a reduced HELIX-RC speedup for most of the benchmarks on 16 cores. Figure 5.2 shows that generally, as the core architecture becomes more capable, the speedup of both the integer and floating point benchmarks decreases (only 256.bzip and 175.vpr have negligible speedup differences between the in-order and out-of-order cores). The speedups here are relative to the single-threaded execution of the same core type. For example, on the 2-way in-order core, the parallelized version of 197.parser is approximately 7× faster than the single-threaded execution on the same core, whereas the parallelized version of the code is only 5.9× faster on the 4-way out-of-order core than the single-threaded execution on the same core.

While the HELIX-RC speedups change across core type, overall performance increases when a more capable core is used. Figure 5.3 shows the performance of HELIX-RC parallelized code on the three different core types relative to the performance on the 2-way in-order core. Regarding the absolute performance of the multithreaded code, the 4-way out-of-order core always has higher performance than the less capable cores. In some cases, the multithreaded performance scales almost

as well as the single-threaded performance—around 2×. However, in some cases, the performance is barely higher than that on the 2-way in-order core; for example, there is only a 20% improvement for 183.equake. While it is encouraging that the significant number of transformations performed by HCCv3 never reduces the extractable ILP for more complex cores, it is important to understand why the multithreaded performance scales worse than single-threaded performance and whether anything can be done to improve it.

### 5.1.2 Speedup Degradation in Out-of-Order Cores

We have seen several reasons why out-of-order cores have lower speedups than in-order ones. Next, I will describe the sources of reduced performance that account for all of the speedup gaps between the in-order and out-of-order cores. This analysis is useful for guiding parallelization techniques that might target even more complex cores in the future. The sources of reduced performance are as follows:

- low ILP sequential segments limit the overall multithreaded performance (164.gzip);
- there is a suboptimal selection of which loops to parallelize (197.parser);
- small loop invocations fail to extract ILP (300.twolf);
- there are memory-bound regions of both single- and multithreaded versions of the code, resulting in Amdahl-bound performance improvement (181.mcf); and
- L1 spatial locality and memory predictability are destroyed in the multithreaded code by HELIX-RC's distribution of loop iterations across cores (177.mesa, 183.equake, 179.art, 188.ammp).

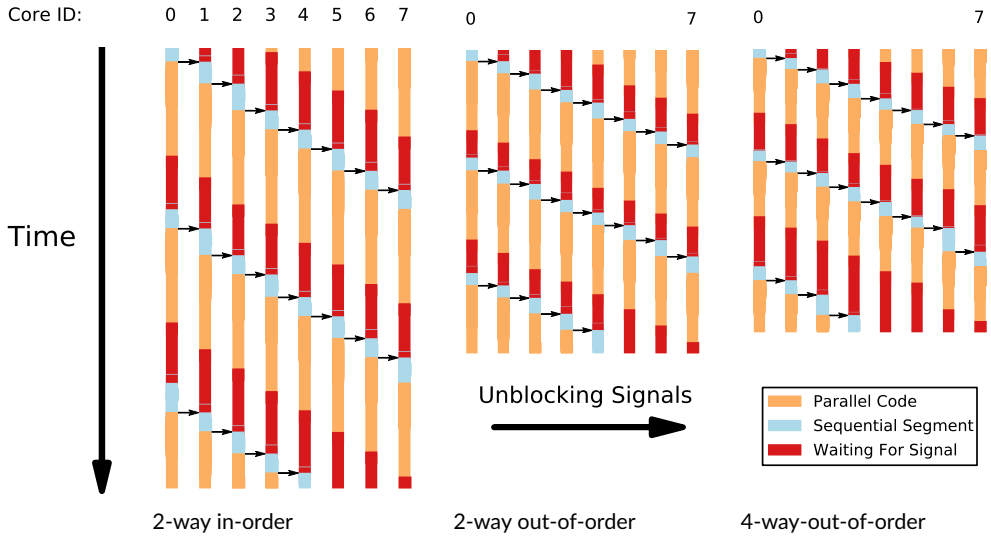



**Figure 5.4:** A single sequential segment is the speedup bottleneck for a loop in 164.gzip (16 iterations of an actual execution trace shown). Out-of-order cores are unable to extract as much ILP from the read-after-write chain in the sequential segment, compared to normal code. Even though execution time decreases, speedups are still lower than those obtained with in-order cores.

### Low ILP in a Sequential Segment

To obtain high speedups, parallelized loops need sufficient parallel code or overlapping sequential segments such that cores are rarely waiting for computation on other cores to finish. While significant portions of computation can be overlapped in many loops, this cannot be done for some loops; this results in the dependence waiting overhead shown earlier in Figure 3.13, which severely limits the speedup of 164.gzip. The loops that limit the performance of 164.gzip on the in-order core are also responsible for the even worse speedup on the out-of-order cores. Most of these 164.gzip loops have the characteristic that there is only a single relevant sequential segment followed by a small amount of parallel code per iteration. The execution time of these loops is set by the time it takes to execute the sequential segment and communicate the signal to the next core, since there is not enough parallel code to cover the waiting time. Figure 5.4 plots the execution time of 16 iterations of one such

loop on each of the three core types. There are 8 vertical bars per core type, one for each simulated core. Each bar represents the state of a core over time. At any point in time, a core can be executing a sequential segment, executing parallel code, or waiting to enter a sequential segment. Figure 5.4 shows that each core has to wait to enter the sequential segment each time it is executed. This produces an overall speedup of around 3× on 8 or more in-order cores.

This sequential segment bottleneck for 164.gzip persists despite the core type; the reason for the speedup discrepancy across core types is that out-of-order execution accelerates vanilla single-threaded execution more than a sequential segment. For the single-threaded execution of the loop shown in Figure 5.4, the 2-way and 4-way out-of-order cores have a speedup of 1.5× and 2×, respectively, over the in-order core. In contrast, the sequential segment has only a 1.4× and a 1.5× speedup for these cores, respectively, and therefore the parallel version of the loop only speeds up by those factors. This is because ILP, which is what out-of-order cores take advantage of, is less in a sequential segment compared to the corresponding original sequential code, as a result of how the HCCv3 compiler generates sequential segments. A sequential segment is often the critical path of execution. Consequently, HCCv3 inserts only the code required to unblock execution of the next core. This results in code that is a chain or tree of read-after-write–dependent instructions—the minimal amount required to calculate the loop-carried dependence. Notice that the latency of executing this dependent code is not drastically improved by a more complex core, since highly dependent code does not benefit as much from multiple issue or out-of-order execution. In contrast, the single-threaded version of the code is not repeatedly blocked by wait instructions that delimit a sequential segment and therefore does not pay the penalty of executing the dependent code after a standstill. The latency of the dependent code is no longer as important, since there are no subsequent cores to unblock, so these instructions are executed concurrently with other instructions in the iteration. This characteristic of sequential segments is intrinsic to the HELIX-RC execution model, since they represent the critical path of execution.

### Suboptimal Loop Selection

The HCCv3 compiler selects the most promising loops to parallelize by examining their estimated performance if parallelized. To estimate their parallel performance, HCCv3 includes a profiler to capture the behavior of the ring cache; it profiles loops on representative inputs. During profiling, instrumentation code emulates execution with the ring cache, resulting in an estimate of the time saved by parallelization. Finally, HCCv3 uses a loop nesting graph annotated with the profiling results to choose the most promising loops.

The emulation significantly simplifies the ring cache and the rest of the platform to enable the compiler to profile code with low overhead to limit the compilation time. This simplification results in estimation errors that are higher for out-of-order cores; I observed that some loops of 197.parser for which the compiler predicts marginal speedups are actually slower on the simulated 16-core platform. When loop selection is improved to eliminate these misestimated loops, speedup increases—in particular for 197.parser, and especially on the out-of-order cores.

### Short Loop Invocations

HELIX-RC's execution model requires *between-loop synchronizations*—which entail overhead—to be performed before and after a parallelized loop is executed. Before a parallelized loop is executed, the initial memory address where the parallelized code is stored needs to be propagated to all cores. To do this, HELIX-RC relies on a special wait/signal pair that is executed just before jumping to a parallelized loop. After a parallelized loop is executed, there is a memory barrier, because values produced in a loop may be read outside the loop. This barrier is implemented with a sequence of waits and signals that instruct each core to flush its local ring node memory before informing the master core that it is safe to resume execution of the code outside the parallelized loop that has been just executed.

As long as the execution time of each loop invocation is large compared to these synchronizations, their overhead can be ignored. But this is not the case for 300.twolf. Aggressive loop splitting, which the compiler performed to improve the amount of parallelism extracted, also resulted in short loop invocations for 300.twolf. Between-loop synchronizations therefore became significant, nearly 10% of the total cycles for one phase of 300.twolf. Additionally, there is so little work to do per core that the Out-of-order cores are unable to extract meaningful ILP. In the single-threaded case, the more complex cores are able to extract significant ILP between loop invocations; hence, the relative speedup on the more complex cores decreases.

Unfortunately, even when the compiler takes this overhead into account while compiling the parallel code, it makes identical loop selection decisions since it is unable to find any loops with better performance. A possible solution to this problem is to reduce the between-loop synchronization overhead by, for example, adding speculation support to allow cores to start executing the code outside a parallelized loop sooner.

### Memory-Bound Workload

181.mcf is relatively memory bound compared to the other SPEC benchmarks [29], as it traverses memory in an unpredictable fashion. Although nearly all of the per-loop speedups on the out-of-order cores are equal to those on the in-order cores, the overall speedups on the former still decrease. This is due to the several phases of 181.mcf that are memory bound to the point that even the single-threaded execution time is nearly identical on all of the examined core types. The result is an Amdahl-bound performance improvement—even though the out-of-order cores are able to accelerate parts of the benchmark, the memory-bound phases limit the overall speedup. Moreover, since the memory-bound region now accounts for a larger portion of the out-of-order cores' execution time, speedup decreases compared to the in-order core speedup. Since this is a characteristic of the benchmark, there is not much the compiler can do to compensate.

### Disrupted Memory Access Predictability and Spatial Locality

The limitation in the speedup for the floating point benchmarks is a result of their worse L1 spatial locality and fewer L1 prefetch successes. This was the case for all of the examined core types and is represented in the memory bar in Figure 3.13. This was also observed for HELIX-UP [10], where significant L1 locality was lost by HELIX loop parallelization. The source of this lost locality is the distribution of iterations across cores. Imagine an array whose $i$th element is accessed on the $i$th iteration of a loop. In the single-threaded case, the L1 will have good spatial locality, as some iterations will be L1 hits by virtue of accessing the same cache line as a previous iteration. With HELIX, however, a core does not execute subsequent iterations—if core 0 executes iteration 0 on a 16-core system, the next iteration it will execute will be iteration 16. Depending on the size and the type of the array, the $i$+16th element will likely not be on the same cache line as the $i$th element, leading to a cache miss. In addition, the larger strides between accesses make it difficult for the prefetcher to detect subsequent accesses, as it is now more likely that subsequent accesses from a single core will cross page boundaries. The simulated prefetcher operates on physical addresses and cannot prefetch across a page. A stride that was large to begin with in the single-threaded case will become 16× as large in the case of HELIX and therefore that much more likely to cross a page boundary.

The lost locality affects parallel performance for all of the examined core types. However, the effect of the lost locality is to make the benchmarks more memory bound. As was the case with `181.mcf`, the more memory bound the benchmarks become, the less utility there is from the more complex cores. Unlike the case of `181.mcf`, however, the increase in memory boundedness only affects the parallel version of the code, so the relative speedup between the three core types is more significant. The more complex cores can execute the single-threaded versions of the code much faster, since locality is not lost, and the prefetcher can take advantage of the fairly regular access patterns in these benchmarks. This makes the difference between the single-threaded performance and parallel

performance on the out-of-order cores more pronounced, resulting in the lower speedups.

While HELIX-RC's loop transformations only hurt out-of-order speedup somewhat, they may hurt it more for even more complex cores. The compiler may need to be modified to take different architectures into better account. Future work must look more carefully at the implications of different architectures, to ensure that HELIX-RC is robust across them.

## 5.2 HELIX-RC vs. Multiprogram Parallelism

HELIX-RC increases program performance by using otherwise idle cores. Another way to increase utilization of idle cores is to run multiple programs concurrently. One might suspect that since HELIX-RC does not scale perfectly linearly with the number of cores, it cannot compete with pure multiple-program parallelism in terms of extracting raw computing throughput. I will show that, counterintuitively, often the automatically parallelized version of a benchmark gets far better use of core resources, not only in terms of single-program performance but also in terms of overall throughput.

Assuming resources are not shared between programs, 16 copies of a program on 16 cores should achieve a throughput that is 16 times that of a single copy. But on modern processors, multiple resources are shared between cores, such as DRAM bandwidth, last-level cache (LLC) storage, prefetching hardware, and on-chip network bandwidth. All of these resources can degrade ideal performance if overburdened. Looking at LLC cache contention, for example, a single 16-threaded HELIX-RC process will only have the working-set footprint of one single-threaded process, whereas 16 copies of the program will have a working set footprint that is 16× larger. In such situations, HELIX-RC excels by greatly reducing the burden on any limited resources whose use scales with the number of running programs.

This section shows that running one or more multithreaded HELIX-RC processes often improves both program latency (i.e., execution time) and throughput if shared hardware resource

contention is considered, compared to running multiple single-threaded program copies. Alternatively, for a fixed throughput target, running HELIX-RC processes can reduce the amount of shared resources that are required. In other cases, where HELIX-RC speedup is not high enough to compete with multiple programs on throughput, Helix-RC compiled programs still present a trade-off between program latency and throughput that can be exploited for situations that may require a balance between single program performance and total system throughput.

### 5.2.1 Experimental Setup

In the SPEC 2000 workloads running on the default 16-core system, I found that the LLC is the only shared resource that degrades the performance of multiprogram execution. The DRAM bandwidth, on the other hand, is more than sufficient for the SPEC 2000 benchmarks, even when 16 copies of a benchmark are running. As a result, the simulation experiments consider only contention for the LLC.

#### Benchmark Classification

Depending on the working set size, a benchmark may or may not improve throughput by increasing the number of processes (either HELIX-RC processes or vanilla single-threaded processes). Therefore, I classify the benchmarks used into different categories in Table 5.1 based on working set size, as obtained from [29] and the present study's simulation results. First, 183.equake has a very large working set of approximately 32 MB, comparable to the LLC sizes of modern server processors. Next, 179.art and 181.mcf have large working sets (~4MB), but not so large as to completely fill a modern LLC. The rest of the benchmarks have decreasing working sets ranging from 1 MB down to less than 1/8th of a MB. For the sake of space and simulation time, I will explore one benchmark from each of the five categories.

**Table 5.1:** Working set sizes for SPECint 2000.

| Category | Working Set Per Process | Benchmarks |
|---|---|---|
| Very large | 32 MB | 183.equake |
| Large | 3-4 MB | 179.art, 181.mcf |
| Medium | 0.5-1.0 MB | 188.ammp, 300.twolf, 175.vpr |
| Small | $<$ 0.5 MB | 197.parser, 256.bzip2 |
| Negligible | $<$ 0.15 MB | 164.gzip, 177.mesa |

### Simulation results

Simulations were run on the default 16 in-order core, 4 DDR3 memory channel system. The LLC size was swept from 4 MB to 32 MB. The original default size, 8 MB, is typical of modern desktop processors, while the largest size, 32 MB, is typical of modern high-end server processors. Only one benchmark at a time was considered.

In addition to considering a single 16-threaded HELIX-RC process versus 16 single-threaded processes, we can also evaluate the performance of running multiple copies of 8-, 4-, or 2-threaded HELIX-RC processes. Consider the HELIX-RC speedups for different core counts in Figure 3.12a, which shows better scaling for lower core counts. For example, on two cores, most of the SPECint benchmarks achieve nearly a 2$\times$ speedup. On four cores, most benchmarks achieve over a 3$\times$ speedup. As the core count continues to increase, the amount of speedup gained becomes less linear with the number of cores. This presents an opportunity to get a closer to ideal throughput improvement by running multiple copies of fewer HELIX-RC threads, at the expense of decreasing the speedup (i.e., increasing the latency) of any individual copy.

When multiple HELIX-RC processes are run at the same time, the 16 cores in the system are statically allocated such that each process has an exclusive set of cores. The ring cache was modified to include address space IDs for memory locations and signals. The ring topology was left unchanged: a single unidirectional ring connects all the cores. This increases the perceived core-to-core commu-

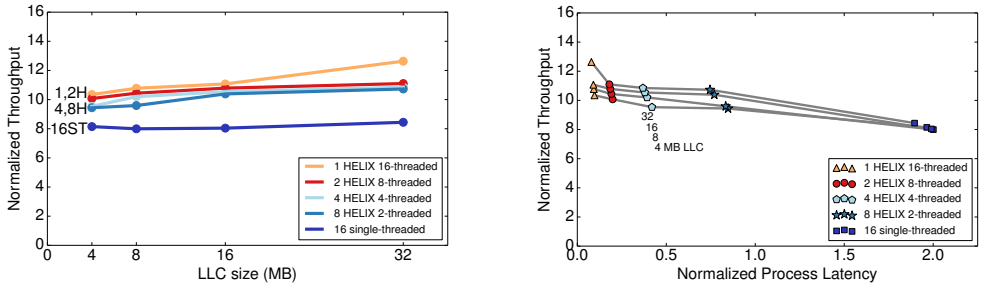



**Figure 5.5:** 183.equake has a 32 MB working set, which results in better performance when fewer processes are used, even on large cache sizes. Only a single 16-thread HELIX-RC process fits in the LLC.

nication latency when multiple HELIX-RC processes are running simultaneously. The total system throughput and process latency (i.e., the maximum execution time of any process) were collected. The throughput and latency are normalized to the case of a single thread running on the default system (8 MB LLC).

### 5.2.2 Evaluation

For benchmarks with large working sets, running multiple HELIX-RC programs sequentially increases program throughput and lowers program latency compared to running multiple sequential programs concurrently. This is because the much smaller working set of one multithreaded process can make better use of a shared LLC, even if the HELIX-RC speedup does not scale linearly with the number of cores. Finally, HELIX-RC enables different throughput/program latency tradeoff points for benchmarks with smaller working sets. These counterintuitive results show that automatically parallelized programs are often a superior option even in cases where abundant multiple-program parallelism is available.

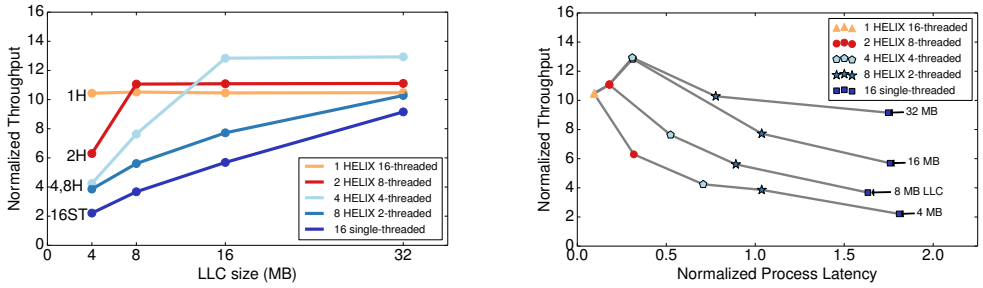



**Figure 5.6:** 179.art's relatively large 4 MB working set means that only 8 processes can fit comfortably at the largest cache size. Depending on the cache size, different HELIX-RC configurations are ideal.

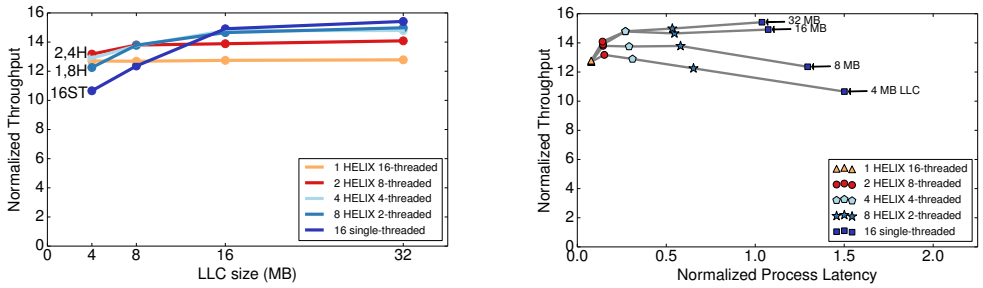



**Figure 5.7:** 188.ammp's medium working set means that the pure multiprogramming case (16 single-threaded processes) outperforms HELIX-RC at large cache sizes.

## Large Working Set

Figure 5.5 shows the results for the benchmark with the largest working set, 183.equake. A single HELIX-RC 16-thread process obtains higher program latency and overall system throughput than any other option. The working set of a single 183.equake is approximately 32 MB, the largest evaluated cache size. For this reason, performance is flat across all configurations, except for a single process running on 32 MB, where it just begins to fit in the LLC. All other configurations are limited by the LLC size, so performance decreases as the number of processes increases, even though HELIX-RC scales better with the core count for those configurations.

The next largest working set benchmarks are 179.art and 181.mcf (4 MB); I chose the former to be

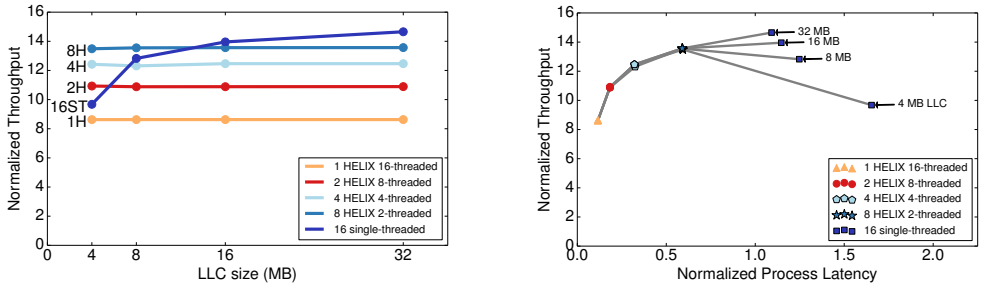



**Figure 5.8:** HELIX-RC gives 197.parser a number of reasonable latency/throughput tradeoff possibilities.

representative. Figure 5.6 shows that different configurations are better in terms of both throughput and latency for different cache sizes. As the LLC size increases from 4 MB to 32 MB, the optimal configuration moves from one HELIX-RC process using every core to four HELIX-RC processes using four cores each. A larger number of processes begins to introduce too much contention for the LLC.

### Medium Working Set

I use 188.ammp to represent medium-sized working set benchmarks. Figure 5.7 is the first example where the non-HELIX-RC multiprogramming case is superior to every HELIX-RC configuration, but only at the largest cache sizes. At an LLC size of 8 MB, a single HELIX-RC process matches the throughput of 16 single-threaded copies, but with the benefit of much better program latency.

### Small Working Set

Figure 5.8 shows 197.parser, and Figure 5.9 shows 164.gzip. For the former, only the 16-process configuration is ever limited by cache size, whereas for the latter, none of the configurations are so limited. For 164.gzip, HELIX-RC never outperforms the vanilla multiprogram, even though multiple smaller HELIX-RC processes can extract more throughput than one 16-core HELIX-RC process. This is primarily the result of 164.gzip's poor scaling beyond two cores: the compiler is unable to

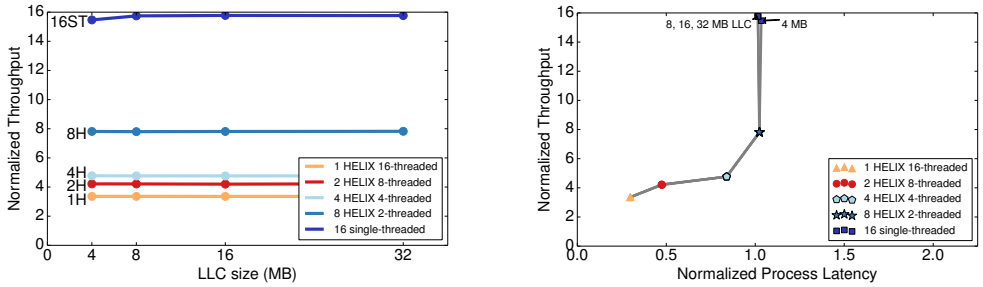



**Figure 5.9:** The worst-scaling HELIX-RC benchmark, and also the most CPU bound, 164.gzip achieves better throughput as the number of threads increases, at the expense of some latency.

extract enough parallelism to make good use of 16 cores. Even though HELIX-RC never excels in terms of throughput, the various configurations still provide multiple points where there is a trade-off between program latency and throughput in scenarios where single program performance matters. Similarly, maximum throughput is extracted from 197.parser, but only when the LLC exceeds 16 MB. Unlike 164.gzip, 197.parser scales better with the number of HELIX-RC threads, so a number of competitive throughput/latency points are available. Additionally, 8 two-threaded HELIX-RC processes with a 4 MB LLC can nearly match the throughput of 16 single-threaded processes with a 16 MB cache, showing that HELIX-RC can enable better performance on more resource-constrained systems.

HELIX-RC provides an interesting opportunity for a tradeoff between program performance and throughput in the presence of shared resources. This initial study shows that relieving pressure on the LLC can produce a significant benefit. This is even more notable since the SPEC benchmarks that are used are many years old; more recent programs may have even larger working sets and might stress other shared resources, such as DRAM. Further work evaluating modern programs on modern platforms is needed to fully understand the role that HELIX-RC could play in optimizing throughput and performance in different scenarios (e.g., desktop, mobile, server) and under different realistic resource constraints.

## 5.3 Potential HELIX-RC Research Opportunities

In this section, I briefly present some other promising lines of research with HELIX-RC.

### 5.3.1 Compiler Engineering Improvements

Due in part to the need for a very accurate memory dependence analysis, it could take one or more days to compile a single benchmark. This slow compilation speed obviously makes it difficult to try out new ideas. A thorough analysis of the time consumed by each compilation step could be useful to target compiler speed improvements. One reason the memory dependence analysis is so slow is that the algorithm iterates until it converges on a solution. I suspect (but do not know) that a large portion of the memory dependence analysis execution time is spent improving the quality (i.e., the accuracy) of the analysis by very little, thus taking a disproportionate amount of time. Large gains in compilation speed could occur if just a small bit of accuracy were sacrificed. Figure 5.10 depicts the expected tradeoff in dependence accuracy as a function of time spent on the analysis. Most of the benefit may be gained in a short amount of time. It will be worth looking into how much HELIX-RC speedups are affected when the analysis time is reduced.

### 5.3.2 Compiler Sweeps

The evaluation of HELIX-RC in Chapter 3 depended on various heuristics to set many of the compilation parameters. Many of these parameters have non-obvious "sweet spots," and without in-depth analysis, it is difficult to tell what their larger system effects are. For instance, the compiler split sequential segments into multiple smaller sequential segments considering only the potential parallelism among segments, and not the side effects of the extra wait and signal instructions that were necessarily added to the code. Figure 5.11 shows a potential relationship between the number of segments and overall speedups—it will not necessarily be easy for the compiler to determine stati-

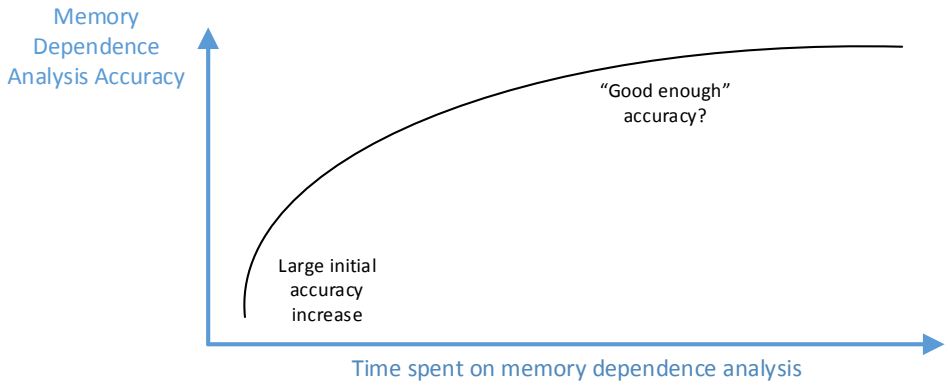



**Figure 5.10:** The memory dependence analysis that HELIX relies on takes a long time to converge on a solution. I suspect that the quality of the analysis may plateau relatively quickly, potentially allowing compilation time to be reduced by ending the analysis early on.

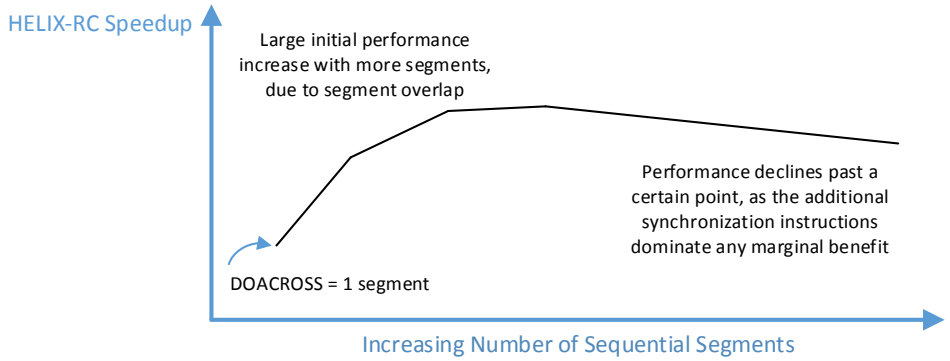



**Figure 5.11:** By splitting the sequential region of a loop iteration into a large number of sequential segments, HELIX potentially improves performance by allowing different segments to be executed in parallel. However, at some point the overhead of the additional synchronization signals will overwhelm any benefit and will also require a larger signal buffer hardware structure.

cally how much splitting it should do. Additionally, many transformations (such as loop unrolling, loop splitting, method inlining) are performed early in the compilation process to make the code more amenable to parallelization (e.g., by simplifying subsequent analysis/transformations). However, as in the case of splitting sequential segments, the exact effects of these transformations are difficult to predict. Excessive loop unrolling and method inlining, for example, can bloat the code size and therefore instruction cache misses. Heuristics were used for these transformations, in part due to the extremely long compilation time as discussed in the previous section. It was therefore not feasible to sweep these parameters to understand their effects. If compilation speed is improved, there will be an opportunity to take a methodical look at the various compilation parameters. By sweeping them and exploring different heuristics, the compilation process can be made more intelligent, which should improve HELIX-RC speedups.

### 5.3.3 Multiple-Loop Execution Model

In the evaluation section of Chapter 3, I showed that HELIX-RC speedups scale relatively well with core count. However, when fewer cores were used, speedups scaled more linearly with fewer cores than with more cores. This is part of the reason why HELIX-RC was able to provide a better performance/throughput tradeoff than normal multiprogramming in Section 5.2. However, this was in the context of multiple HELIX-RC processes. Ideally, a single HELIX-RC process would be able to run multiple loops in parallel, each using only a subset of the available cores, thus boosting single-threaded performance more than using all the cores on a single loop. However, it is very difficult to determine which loops can run in parallel. Although the memory dependence analysis within a small loop can be very accurate, the accuracy of such an analysis *between* small loops is probably much lower. Because the scope of the analysis is larger, there will likely be a large increase in apparent dependences. Therefore, it is very likely that speculation will be required to run multiple loops in parallel.

The hardware necessary for multiple loops may be difficult to create. Since values may need to be forwarded between executing loops, a single ring network would not be sufficient. Instead, some other topology, such as a 2D mesh network, could be used in a circuit-switched fashion. Multiple arbitrary groups of rings could be set up and torn down dynamically with every loop invocation. Special circuits between the rings could be used for inter-loop dependences. However, if speculation is necessary, this will require significant hardware changes, as in TRIPS, Multiscalar, and Hydra, for example.

# 6

# Conclusion

Despite much effort expended in the past, automatic parallelization of irregular workloads remained out of reach. This dissertation has proposed a compiler–architecture co-design that unlocks the previously inaccessible parallelism possible for small loops. This robust solution, HELIX-RC, produces speedups of 6.85× for hard-to-parallelize programs. Moreover, little additional hardware is required, and there is a very well-defined, simple interface between it and a core's existing memory hierarchy. Through the addition of this hardware, automatically parallelized irregular programs can make much better use of multiple cores, in terms of performance and throughput, than even the "easy" parallelism provided by multiprogramming.

# A

# Ring Cache Technical Report

## A.1 Introduction

In ISCA 2014, [9] we proposed HELIX-RC, a compiler–architecture co-design for automatic parallelization of irregular programs. The combination of an automatically parallellizing compiler and a custom-designed piece of hardware logic demonstrated a nearly 6.85× speedup on unmodified, highly irregular SPECint 2000 benchmarks. The success of HELIX-RC emanates from a new piece of hardware, the *ring cache*, which helps overcome the primary bottlenecks in HELIX-style parallelization: data communication latency and sequential forwarding synchronization chains. Without ring cache, the HELIX compiler was limited to producing around a 2× speedup on commodity multicore chips [11].

In the HELIX-RC paper, the technique was evaluated on a C/C++ based x86 simulator called XIOSIM [33], with the ring cache similarly modeled in C++. While every effort was made to model the ring cache at a cycle-accurate level, high-level languages are not the best fit for expressing hardware operations. To increase confidence in the ring cache, this report presents a fully tested and synthesizable Verilog reference design, as well as fully detailed explanations of its implementation and our design decisions. Due to the constrained length of conference proceedings, the HELIX-RC paper only explored the high-level details of the ring cache. In contrast, the goal of the present report is to fully flesh out any missed details so that our results can be replicated, and so that other researchers, with the aid of our Verilog, can explore and evaluate the ring cache in FPGAs or even silicon.

Before proceeding, the reader should be familiar with the HELIX-RC paper, in order to have an appreciation of HELIX-style parallelization and at least a high-level overview of how ring cache operates. The rest of this report is organized as follows. First, a brief summary of HELIX-style parallelization is presented. Next, a high-level overview of ring cache and its components is given, to establish terminology and to place more detailed explanations in context. Then implementation details and design decisions for these same core components are presented, with datapath schematics, control FSMs, and timing examples provided where appropriate. Finally, we describe some preliminary synthesis results for our reference design. The full Verilog implementation is included as a second appendix .

## A.2 Background

While it is assumed that the reader of this report has previously read the HELIX-RC paper [9], this section provides a limited background for HELIX and ring cache. This review of certain aspects of HELIX-RC serves as a reference point for descriptions and design decisions discussed later in the implementation section of this report. For the full background for HELIX, the original CGO

paper [11] and subsequent ISCA paper [9] should be consulted. In this section, we describe the basic HELIX execution model and the method through which HELIX extracts parallelism from single-threaded code. Further, we highlight the main performance bottlenecks that HELIX suffers on a commodity multicore system, mostly data communication latency and limitations resulting from sequential forwarding synchronization chains. Finally, we describe how ring cache alleviates these bottlenecks by decoupling data forwarding from data generation and decoupling signal forwarding from synchronization. These concepts set the framework for understanding the motivation behind many of our ring cache design decisions.

### A.2.1 HELIX Execution Model

HELIX achieves a speedup of single-threaded code by automatically parallelizing loops that run on multiple cores of a single chip. The compiler performs extensive memory dependence analysis and code transformations to determine which loops it should parallelize in a program, and it chooses these loops at compile time. HELIX executes loops in parallel by assigning loop iterations to cores, with subsequent iterations of a loop assigned to subsequent cores. Specifically, in an $N$-core system, core $i$ first executes iteration $i \bmod N$, then iteration $N + (i \bmod N)$, then iteration $2N + (i \bmod N)$, and so forth. This mapping produces a unidirectional flow of shared data and synchronization signals from past iterations to future iterations in a logical ring of cores. Figure A.1 depicts an execution timeline for HELIX parallelized code. During a program's execution, a single core is assigned to be the master core, which executes all code outside of the chosen parallelized loops. When this core reaches the start of a parallelized loop, it indicates to all other participating cores that a loop is about to begin. All cores, including the master core, then jump to the parallel loop. When the loop has finished executing, a memory barrier is executed to ensure that any memory addresses written during the loop are properly visible to every other core. After this memory barrier has been executed, the master core is free to resume executing code between parallel loops, with the knowledge that it is safe

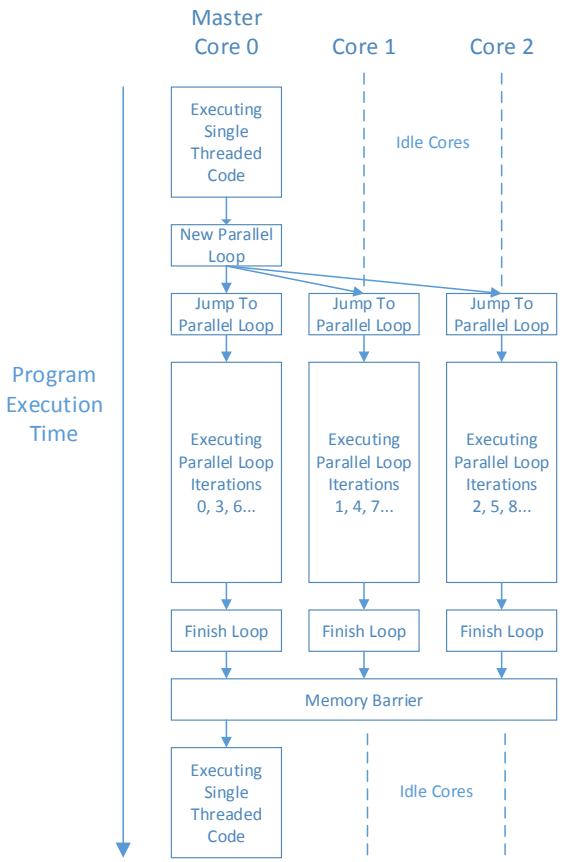



**Figure A.1:** A single core executes the code between parallelized loops, and before executing a parallelized loop, it also instructs other cores to jump to the same loop.

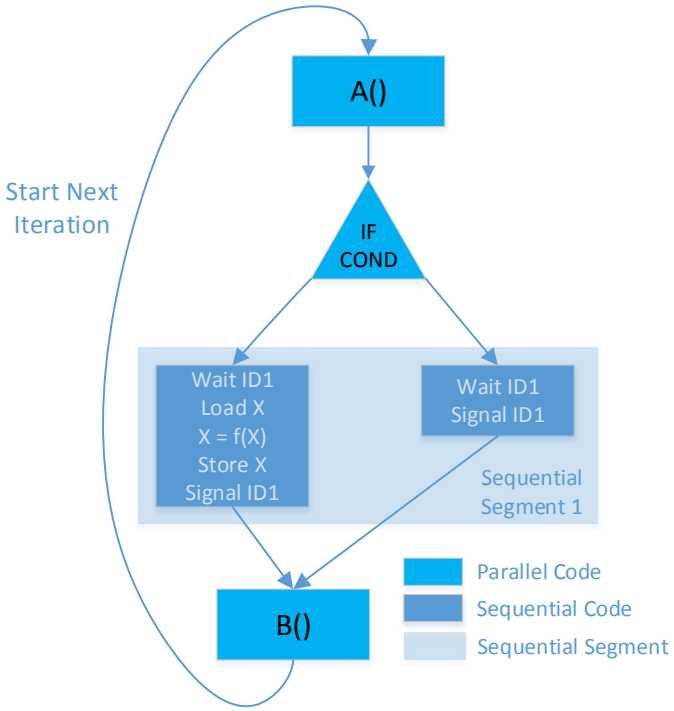



**Figure A.2:** A simplified example of the static code of a loop in 175.vpr containing a single sequential segment. The right branch of the if-statement must still be synchronized, even though no shared data is accessed.

to read or write any memory address.

### A.2.2 Parallel Code

Figure A.2 represents the static code of a loop after parallelization. This loop is roughly representative of a loop in 175.vpr, one of the benchmarks we evaluated. After in-depth memory dependence analysis, HELIX marks the regions of a loop body that access data that is shared across loop iterations (or, at least, data it can't prove is not shared). We call these regions *sequential segments*. At the beginning of each sequential segment, HELIX inserts a `wait` operation that contains a particular `sequential segment ID`. At the end of each sequential segment, HELIX inserts a *signal* operation with the same ID. A `wait` operation prevents a core from entering a sequential segment

with that specific segment ID until the corresponding `signal` has been received from the previous iteration of the loop, which is running on a different core. Therefore, each segment is executed in loop iteration order. This creates a sequential chain of synchronization, as each core must explicitly unblock the iteration running on the next core.

Once a core enters a sequential segment, the loads and stores it performs may involve shared data. It is unknown at compile time which specific addresses will be shared and which other iterations will access those addresses; it is only known that any accesses in that segment might be to shared data. Only code that might access shared data is placed inside a sequential segment—if the compiler determines that some code will never access shared data, it is not placed within a sequential segment and therefore can run in parallel across all iterations. Figure A.2 depicts a loop body with only one sequential segment. If the compiler is 100% confident that certain accesses are independent from others, it will split the sequential segment into multiple smaller segments, which can run in parallel with one another. Even though shared data may be accessed in these multiple segments, the compiler has determined that each segment will access unique sets of shared data.

### A.2.3 Decoupling Data Communication

To illustrate the data communication latency encountered by HELIX on a traditional multicore chip, Figure A.3 (left) shows an execution timeline for a two-core system. At the start of execution, core 0 has entered the sequential segment, while core 1 waits to enter. During the execution of the sequential segment, core 0 stores a value to the address of variable X, whose cache line will be loaded into its L1 cache. The core then leaves the sequential segment by issuing a *signal* to unblock core 1. After some communication latency, core 1 receives the signal and enters the sequential segment. Subsequently, core 1 issues a load to the address of variable X. Because the recently written value of X resides in core 0's L1 cache, there is a cache coherence delay (75–210 cycles on modern Intel CPUs) before core 1 receives the data. Since sequential segments are executed in loop iteration order, the

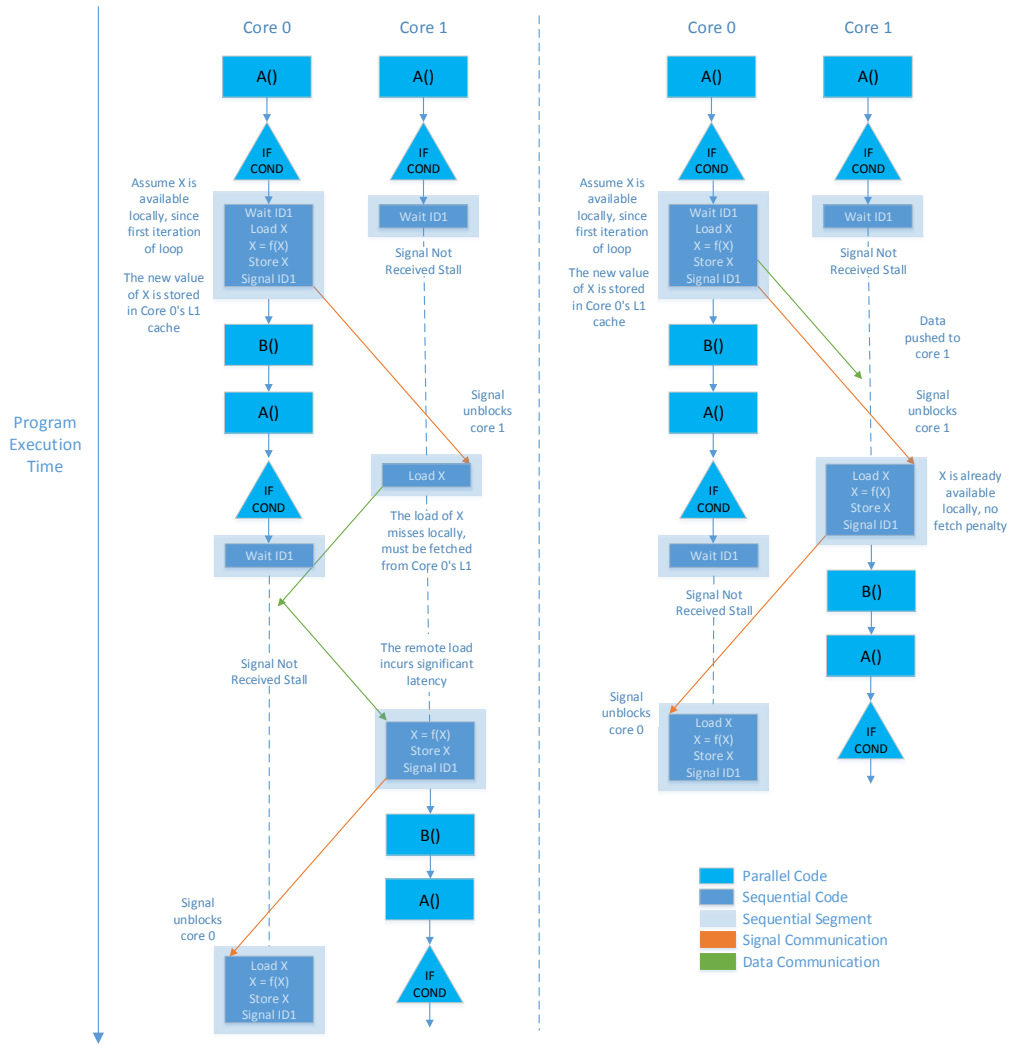
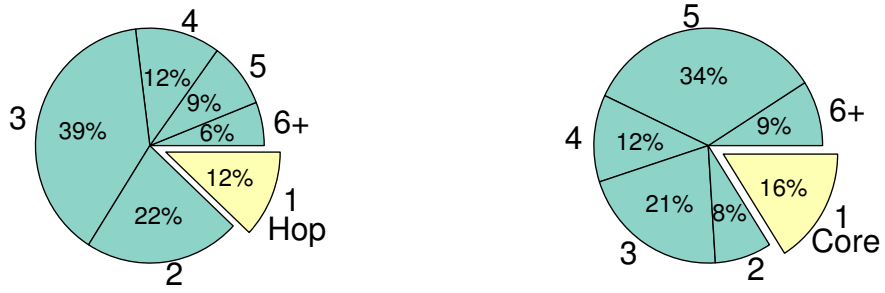



**Figure A.3:** Left: a reactive data communication mechanism results in core 1 stalling on a load. Right: the proactive ring cache reduces the stall by sending the data as soon as it is produced.




Distance Between Producer and Consumer

Number of Consumers

**Figure A.4:** It is very hard to predict which core will need a particular piece of shared data (left). Different pieces of shared data may also have very different numbers of consumers (right). As a result, it is very hard for a core to know which addresses it should prefetch, and when.

data transfer latency significantly increases the loop's critical execution path. This data communication cost greatly restricts the loops that the HELIX compiler can target, limiting it to select loops that have few dependences. The cost is a direct result of the *reactive* nature of cache coherence protocols: data is only moved when it is requested. This produces a coupling effect between the communication of the shared data and its usage.

One reasonable solution would be to aggressively prefetch data. In fact, the original HELIX work [11] used Intel Hyperthreading to prefetch signals, thus reducing their perceived communication latency. When compiling for a traditional multicore processor, HELIX implements `wait`s and `signal`s with vanilla loads and stores, respectively, to special memory locations. A special hyperthread running on each core constantly executes the load corresponding to the wait instruction of the next sequential segment to be executed, so the signal from the predecessor core is effectively pulled locally soon after it is generated, potentially before the subsequent core actually needs it. Without such prefetching, the signal in Figure A.3 (left) would experience a reactive transfer delay similar to the data load.

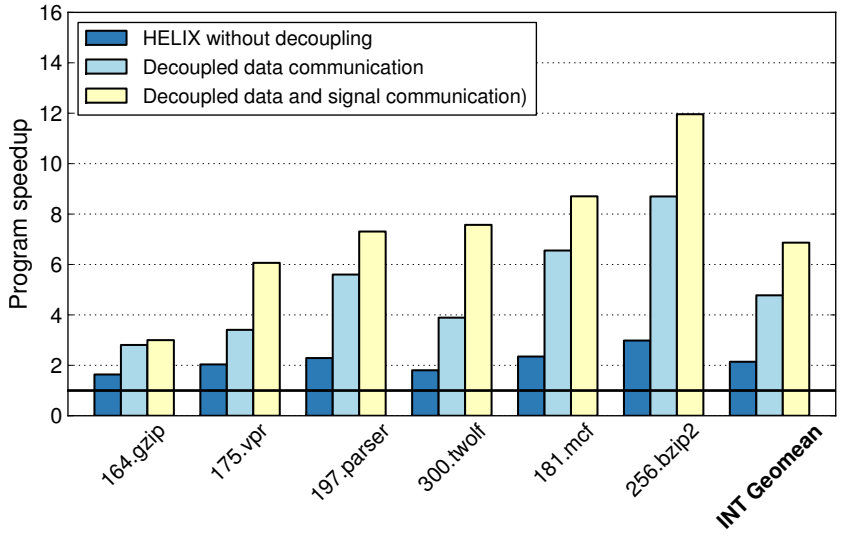



**Figure A.5:** Decoupling data and signal communication with a ring cache drastically improves speedups over vanilla HELIX on a traditional multicore processor.

Prefetching of signals is only possible, however, because the sequence of sequential segments is very predictable. In contrast, shared data is not predictable: variable X in one iteration of the loop may be different than in the subsequent iteration. A good example of such behavior is a traversal of a linked list/tree/graph, where each iteration of a loop follows one or more pointers before performing some work on the corresponding node(s). The memory addresses resulting from traversing such a structure are very unpredictable and there would therefore be no feasible way to prefetch the nodes at those addresses. Moreover, not only is the exact location of the shared data unpredictable, but the number of different cores that access a piece of shared data is also unpredictable. Figure A.4 shows the number of consumers of a given piece of shared data and the distance between the consumers in a logical ring for the SPECint 2000 benchmarks we evaluated in [9]. Because of this unpredictability, proper prefetching is a very difficult problem to solve.

Instead, HELIX-RC ameliorates data communication costs through the addition of ring cache. Rather than *reactive* data communication, the ring cache facilitates *proactive* communication. As

soon as a piece of potentially shared data is produced, rather than being stored in the traditional cache hierarchy, it is proactively distributed to the ring cache in every core throughout the system. As a result, when core 1 tries to access the shared data, it is already present locally and thus there is no data communication cost, as depicted in Figure A.3 (right). Unlike in typical cache coherence protocols, the communication of the data has been decoupled from its consumption. Additionally, since loads, stores, and signals sent to the ring cache do not need to incur any cache coherence protocol overhead, they can execute and propagate very quickly, with low single-digit latencies. Figure A.5 shows that decoupling data communication increases HELIX speedups by more than double compared to using only a traditional cache coherence protocol. Part of this improvement is due to the proactive nature of the ring cache, and part of it is due to the very fast communication.

### A.2.4 Decoupling Signal Forwarding

While decoupling data transfer reduces the perceived data communication cost, there is another decoupling opportunity that ring cache exploits. Consider Figure A.2 once again. The sequential segment in this loop only sometimes accesses shared data, depending on the preceding outcome of the "if" evaluation—we call the right branch of the "if" an empty sequential segment. Traditionally, on a normal multicore processor, HELIX must still synchronize this sequential segment despite the fact that it does not always access shared data. This requirement of executing sequential segments in loop iteration order creates a sequential forwarding synchronization chain, even if it is unnecessary. In the three-core system of Figure A.6, core 1 unnecessarily waits for core 0 to finish executing sequential segment 1 and send the corresponding signal before entering the sequential segment itself. Ring cache breaks this synchronization chain by performing signal buffering. It does this by having each core record how many sequential segments of a particular segment ID it has executed, relative to every other core. This allows each core to correctly decide whether it must execute the `wait` instruction of any particular empty sequential segment or whether it can skip it and continue with

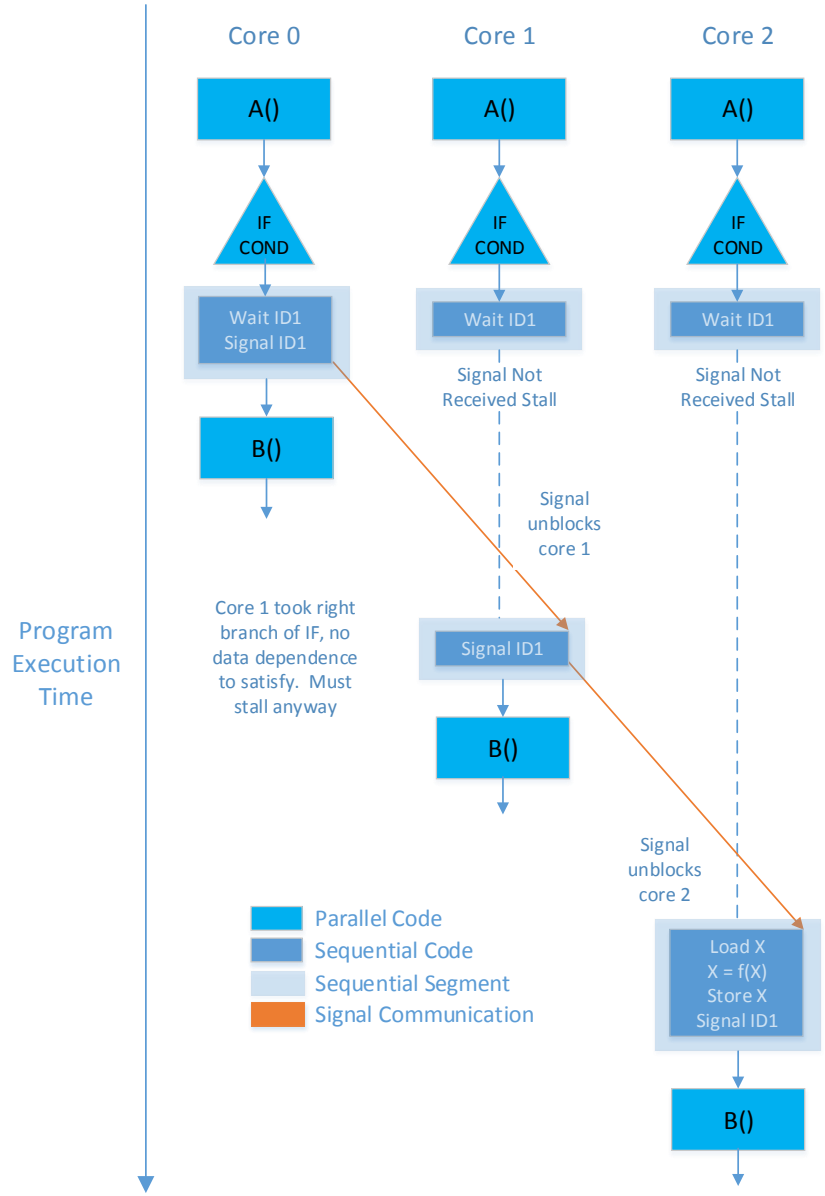



**Figure A.6:** Core 1 needs to stall on the `wait` instruction even though the sequential segment it is executing lacks a shared data access. This creates a sequential forwarding chain between the three cores.

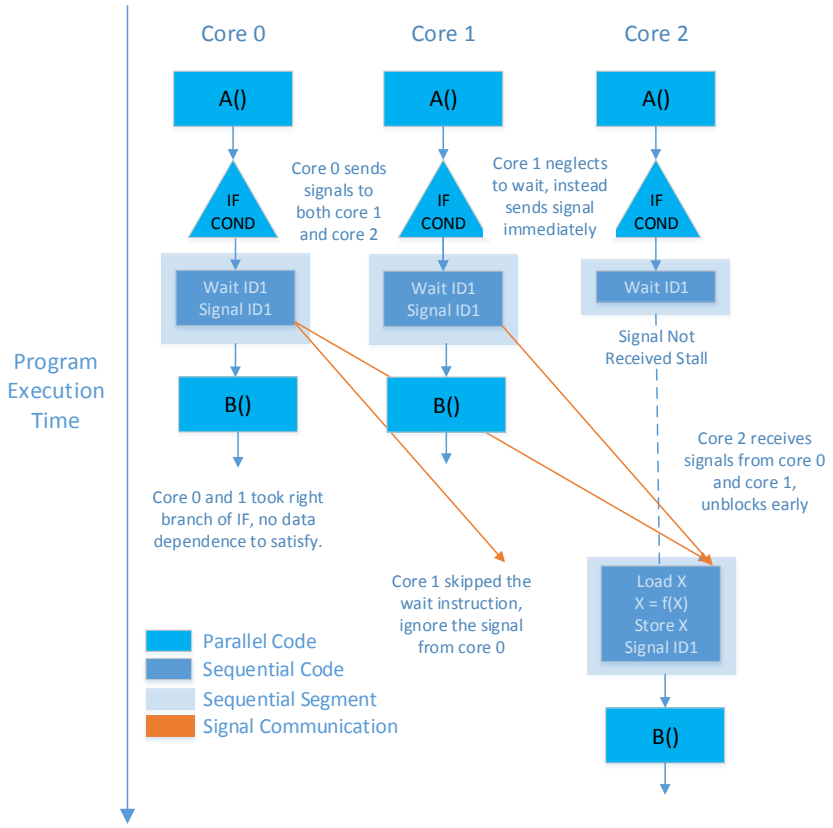



**Figure A.7:** Core 0 simultaneously sends `signals` to both core 1 and core 2. Core 1 skips its `wait` instruction because the sequential segment it is executing is empty and lacks any shared data accesses. By virtue of allowing early signal transmission when synchronization is unnecessary (hence decoupling signal transmission from synchronization) core 2 is unblocked sooner than it otherwise would be.

execution.

This does have the downside, however, of requiring cores to send signals not only to their neighbor in the ring (the subsequent iteration), but to every other core in the ring. Likewise, all cores must now track whether they've received signals from every other core, rather than just from their predecessor (the previous iteration). This makes decoupling signal transmission prohibitively expensive without ring cache, since the additional signals between cores would result in a large increase in cache coherence traffic. With ring cache's hardware signal-buffering capability, however, the benefit outweighs the cost. Some loops benefit significantly, especially if the branch outcome before such a segment changes frequently, as it does in the corresponding loop in 175.vpr. Figure A.7 depicts the result of breaking the synchronization chain. Despite the additional required signals, core 2 is unblocked sooner than it otherwise would be, getting a head start on executing the sequential segment. To better illustrate the decoupling effect, this figure assumes that core 0 can send a signal to both core 1 and core 2 in approximately the same amount of time—and in the ring cache, this is close to being true, since the latency between cores is only one cycle, but the initial time to get from the core to the ring cache may be longer. The important point is that there is no longer a signal chain connecting core 0 to core 1 to core 2. Instead, the presence of the ring cache and the empty sequential segment allows core 0 and core 1 to send signals as soon as they correctly can, rather than being tightly coupled in a forwarding chain. Figure A.5 summarizes the performance improvements HELIX-RC obtains from decoupling both data and signal communication—decoupling both of these increases HELIX speedups nearly 3× compared to systems that lack ring cache. Both decoupling bars use fast proactive communication, and therefore the contribution from decoupling signal forwarding is purely due to breaking sequential chains, not from fast signal propagation, which is already included in the second bar.

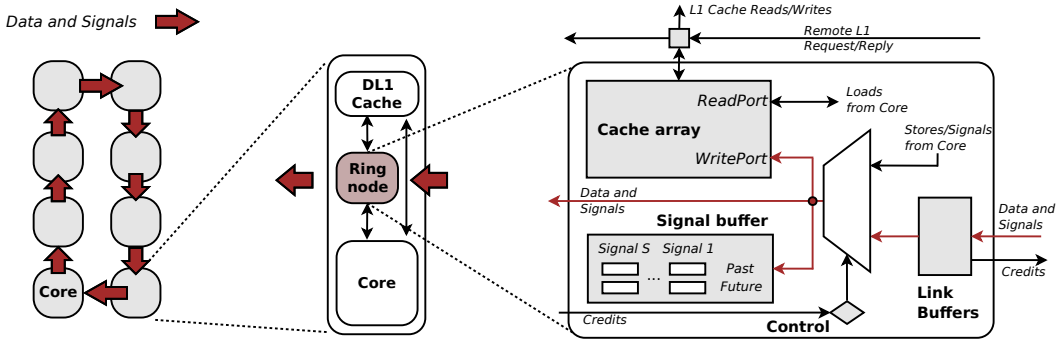



**Figure A.8:** Ring cache architecture overview. From left to right: overall system; single core slice; ring node internal structure.

## A.3 Ring Cache Overview

To establish terminology and a high-level understanding of ring cache, this section presents an overview of its architecture and design. With this basic background, implementation details and design decisions will be more easily understood. This section is divided into three parts: first, the interaction between the core and the ring cache is detailed, followed by the connections between ring nodes, and finally the integration between the ring cache and the rest of the memory hierarchy. In subsequent sections, each of the presented blocks and functions of ring cache will be thoroughly examined at the hardware level.

Figure A.8, from the original HELIX-RC paper, depicts the general structure of a multicore system with ring cache. The leftmost diagram shows the flow of data and signals between cores. Each core has an attached ring node, and these are organized in a ring. The middle diagram shows the contents of a ring node, and the rightmost diagram depicts a high-level view of the ring nodes' internal structure. The ring nodes contain a core interface, a cache array, a signal buffer, a connection to the local L1 cache, and three unidirectional ring connections between nodes. The following subsections will detail at a high level how these different elements are used during a typical core interaction.

### A.3.1 Core–Node Interaction

Every core in a HELIX-RC enabled chip has a connection to and from a private ring node. This connection is used exclusively when a core is executing a sequential segment. If a particular loop is outside of a sequential segment (or lacks sequential segments altogether), then the normal cache hierarchy is used, and the ring node connection is irrelevant.

When a core approaches a sequential segment, it first executes a `wait` instruction, which has an associated sequential segment ID. When ring cache is present in a system, a `wait` instruction is not just a special load instruction but a unique instruction used only by the ring cache. When the `wait` instruction is executed, it is sent to the ring node, along with its ID. The ring node checks its local signal buffer to determine whether enough signals have been received by every other participating core to allow entry into the sequential segment. To enter a particular sequential segment where shared data is accessed (as opposed to an empty one, as discussed in Section A.2.4), a core must have received signals with the correct ID from every other previous iteration of the loop (and therefore every other core). If a sequential segment is empty, a core can enter it as long as sending the associated signal won't overwhelm any other core's signal-buffering capability. If a core cannot yet enter a sequential segment, the `wait` instruction is held by the ring node, and the core stalls. Once enough signals have been received, the ring node releases the `wait` instruction back to the core, which may then proceed to execute the sequential segment.

Once inside a sequential segment, any loads or stores executed by the core must be sent to the ring node for processing. When a load instruction is presented to the ring node, it first searches its local ring cache array for the value. If it is not present, the ring code can utilize the request network and the reply network to find the correct value from other ring nodes. In some circumstances, such as in the case of a cold miss, the value will be loaded from the traditional cache hierarchy, subject to certain rules to maintain memory consistency, as detailed in the following two subsections. Since a

ring node is guaranteed to service a load either locally or through the request/reply networks or the traditional cache hierarchy, all loads appear to hit in the ring cache from the core's point of view, albeit with potentially unpredictable latency. In the common case of a local ring node hit, the correct value is returned in one clock cycle.

Usually, after entering a segment and issuing a load instruction, a core will perform some arithmetic processing on the data before storing an updated value to the ring node. The address and value of the store are handed to the ring node, which immediately writes this value to the local ring cache array. Additionally, the ring node inserts the address, value, and originating core ID into a bundle that is sent over the forwarding network. The forwarding network propagates this information to every other ring node in the ring, one hop per cycle, and each of these in turn stores the value in its local ring cache array. The originating core ID is used to stop propagation of the store once it has circled the entire ring. As a result, every ring node contains the newly updated shared value.

Within this sequential segment, the executing core is the only one who can possibly read or write any shared address belonging to the segment, as guaranteed by the compiler. In order to leave the sequential segment and inform other cores that they may now enter the segment and access said data, the core injects a `signal` instruction to the ring node. Upon entering the ring node, the signal updates the local signal buffer to record the fact that the core is leaving a sequential segment (and therefore can "forget" the signals that granted it entry). Simultaneously, the signal is also added to a bundle and propagated throughout the forwarding network, just like stores. It continues propagating around the ring until it reaches the predecessor of the core that executed the signal, updating all the ring node signal buffers along the way. After the core injects the signal, it has left the sequential segment, and all future loads and stores go to the traditional cache hierarchy, until the next sequential segment is encountered.

Since stores and signals have global effects, the core can only send them to the ring node in program order once they are no longer speculative. Additionally, loads and stores cannot be reordered

around either `wait` or `signal` instructions. In our simulations, we reuse logic from the load–store queues for memory disambiguation to block items in the load queue until the `wait` instruction has returned from the ring node.

### A.3.2 Node to Node Connection

The ring cache implements three different networks: the forwarding network, to propagate signals and data throughout the ring, the request network, to handle loads that missed in a local ring node and are therefore sent to other cores, and the reply network, which returns values requested over the request network. The latter two networks are used infrequently but are necessary for memory consistency. If they are not used infrequently, performance will suffer, since they essentially revert the ring cache's proactive data communication mechanism to a reactive communication mechanism, resulting in a performance penalty similar to that in a traditional cache coherence protocol.

Each network has its own network receive buffers and credit-based flow control. The separation of the different traffic types into three different networks simplifies deadlock prevention, as there are easily understood interactions between the networks. Even with just a single network, deadlock avoidance is a significant concern, owing to the ring topology of the ring cache. To prevent deadlock within a single network, eventual forward progress must be guaranteed. The primary way we ensure this is by using a minimum of two slots per buffer for each network and enforcing the invariant that a new item may not be injected into the network if there is already an item in the corresponding node's receive buffers. This guarantees that there is always an open buffer somewhere in the network, and as long as there is always an open buffer somewhere in each network, some item will always be able to advance. In turn, if some item can always advance, items will eventually proceed to the point where they can leave the network. In the case of the forwarding network, for example, items disappear from the network once they have traversed the ring, so as long as items can continue circulating, they will eventually leave the network and free up slots for other stores/signals.

Although the forwarding network can stall due to evictions from the ring cache array, it will always resume after the eviction to the L1 is complete.

To avoid deadlock, there must also not be any circular dependency between the three networks. The forwarding network never interacts with the other two networks, so it cannot create such a dependence. The request network receives input from a ring node's corresponding core only when it issues a load that misses locally. Items exit the request network at a remote node, where they wait to access the ring cache array. After performing the load (and potentially an L1 request), they enter the reply network, which returns them to their originating core. Since there is a unidirectional flow of data from core to request network to reply network to originating core, and a single core may only be executing one load at a time, the networks cannot deadlock.

In the forwarding network, signals and stores are packaged in bundles and move around the ring in lockstep. For correctness, signals may never pass the stores, since they could accidentally allow a core access to shared data before the data has arrived. By sufficiently provisioning the bandwidth of the ring cache array and signal buffer, bundles can traverse a ring node in a single cycle, since they access the memory/signal buffer in parallel with link traversal. Traversing the entire ring therefore takes the same number of cycles as the number of cores.

The items in the request network are addresses that a core has requested to load and the core ID that requested it. The items in the reply network are the requested data and the ID of the core that originally requested it. Items in these networks can similarly hop between ring nodes in a single cycle. The cost of a remote load is therefore approximately the same number of cycles as the number of cores, plus the time it takes to exit the request network, perform the load to a cache array / L1 cache, and enter the reply network.

### A.3.3 Memory Hierarchy Integration

The ring cache can be thought of as another layer of the memory hierarchy that sits just below the L1 but is exclusively used for shared data in sequential segments. As such, it must maintain any memory consistency guarantees in its interactions with the existing cache hierarchy. This is accomplished through three different invariants. First, shared memory can only be accessed within sequential segments through the ring cache (this is by the definition of a sequential segment, and therefore enforced by the compiler). Second, only a single "owner" core can access a particular shared memory location through the L1 cache on a ring cache miss. Finally, the existing cache coherence mechanism must guarantee total store ordering (as Intel's does), which means that the memory will process stores to a particular address from a particular cache in the order that it receives them.

Interactions between the ring cache and L1 can occur when data is evicted from the ring cache and when a load misses in the ring cache. In the case of evictions, the evicted data will be written back to the L1 only if the owner of a particular memory address is the one performing the eviction—all other cores just overwrite the data with whatever store triggered the eviction. Likewise, when a load misses in the ring cache, a request is sent to the request network to fetch the data from the owner core's ring cache array and, if it is not present there, from the owner core's L1 cache. It is crucial that cache lines are assigned separate owners, or different cores will risk updating the same cache line at the same time. By ensuring that all loads and stores to particular cache lines go through a single core's L1 cache, races to update a particular cache line are avoided, and sequential, in-order memory accesses to that cache line are guaranteed. Having different cache lines have different owners also ensures that the existing cache coherence mechanism won't "ping pong" a cache line containing shared data back and forth between different cores, incurring a performance penalty.

Finally, when a loop invocation is ending, all cores must update their L1s with any shared data that they are the owner of in their local ring cache arrays, since after the loop is over, the core execut-

ing the code between loops can access any memory location. This is effectively a distributed memory barrier.

## A.4 Ring Cache Implementation

The remainder of this report discusses the implementation details of our reference design, including datapath schematics and control FSMs when appropriate. The previous section has established some base terminology and a high level description of the functionality of ring cache. In the following sections, the exact details and in-depth explanations for different functional blocks are described. The description is split into several sections, corresponding to major aspects of ring cache. First, an overall view of the ring cache datapath is presented. Second, the precise interfaces between cores and ring nodes, ring nodes and L1 caches, and ring nodes and other ring nodes are discussed. Hypothetically, those interfaces are all that is needed to integrate our reference design into a system. Then, a description of the end of loop ring cache array flush is detailed. Next, we present details of the three networks that enable data and signal communication between ring nodes. Finally, the primary components that form the core functionality of ring cache – the memory module and the signal buffer module – are described in detail. The exact manner in which the memory handles incoming loads, stores, evictions, L1 accesses, and end of loop flushes is included in this description, as well as an optimization to reduce unnecessary L1 accesses. We also present a fleshed out explanation of how the signal buffer decouples signal forwarding from synchronization, and how the hardware design facilitates this decoupling.

The schematics in the following sections are meant to highlight the most crucial parts of the implementation, but may not contain every last detail of the control logic, for example. The Verilog implementation included in the appendix of this report contains all of the precise details, bitwidths, etc., and likely should be consulted along with the written description to resolve any ambiguity.

## A.5 Datapath Overview

Figure A.9 presents the top level schematic of a ring node. The vast majority of the datapath and control is contained within other major modules. This schematic is meant to serve as a reference for the different top level module connectivity – future sections describe each module in detail. The minor logic that is present (credit registers primarily) will be described in Section A.7. Clock and reset routing is not shown to reduce clutter, but all modules that have state have clk and reset inputs on their respective block. Any blocks without clk and reset inputs are entirely combinational. The major blocks:

Receive Buffers These buffers capture circulating bundles from the three different networks (forwarding, request, and reply). Each buffer contains at least two slots internally, as required to prevent network deadlock and to cover buffer turnaround time.

Load Unit This module arbitrates between loads injected by the core and loads circulating in the request network. It passes the selected load operation to the memory module. After the memory responds to a load, the load unit processes it appropriately, depending on whether it hit or miss locally. If it hit, and the load originated from the local core, the loaded data is returned to the core. If it hit, and the load originated from the request network, the response is injected into the reply network. If the load missed, a remote load is injected into the request network. The load unit is also responsible for forwarding existing items from the request and reply network receive buffers to the outgoing links, where they are propagated to the next ring node.

Bundleizer The bundleizer arbitrates between stores/signals injected by the core, and stores/signals already circulating on the forwarding network. The stores/signals output from the bundleizer are sent in parallel to the memory module (for stores), the signal buffer (for signals), and stopper

module (for both). This module ensures that stores and signals circulate in-order by "bundling" them together into a single structure, forcing them to propagate around the ring in lockstep.

Stopper The stopper module removes stores/signals from the forwarding network if they have completed propagating around the entire ring. Any items left in the network bundle after this pruning are sent over the forwarding network links to the next ring node.

Memory The memory module processes has two logical ports, one for loads and one for stores. Loads are initiated from the load unit, and stores from the bundleizer. Load results are returned to the load unit. When necessary, the memory module writes evicted data to the local L1 cache, and loads required data from the L1 cache. When instructed by the signal buffer, the memory module flushes all stored data to the L1.

Signal Buffer The signal buffer reads the signals output from the bundleizer, and records them internally. When the core executes a wait instruction, the signal buffer is responsible for deciding whether it can be released, or if the core must stall. In the case of a special flush-related wait instruction, the signal buffer also instructs the memory module to begin flushing after the wait is released.

## A.6 External Interfaces

This section documents all of the necessary interfaces between a ring node and its core, in addition to a ring node and its L1 cache. Figure A.10 depicts the signals that connect a ring node, a core, and an L1 cache.

### A.6.1 Core Interface

There are five main inputs between a core and its ring node, in addition to *clk* and *reset*. They are related to the particular instruction that the core is presenting to the ring node for execution. There

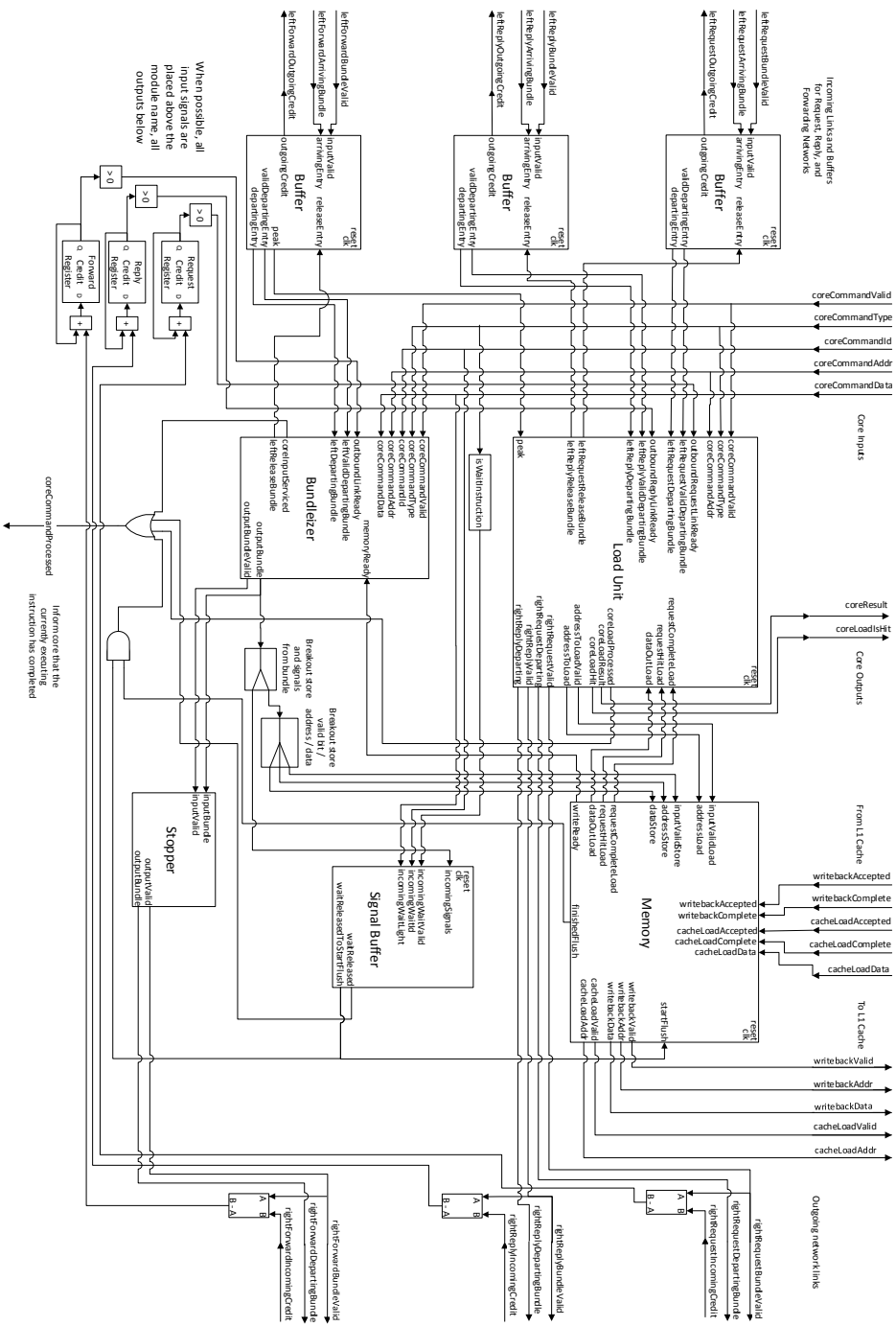



**Figure A.9:** Schematic of top level ring cache module.

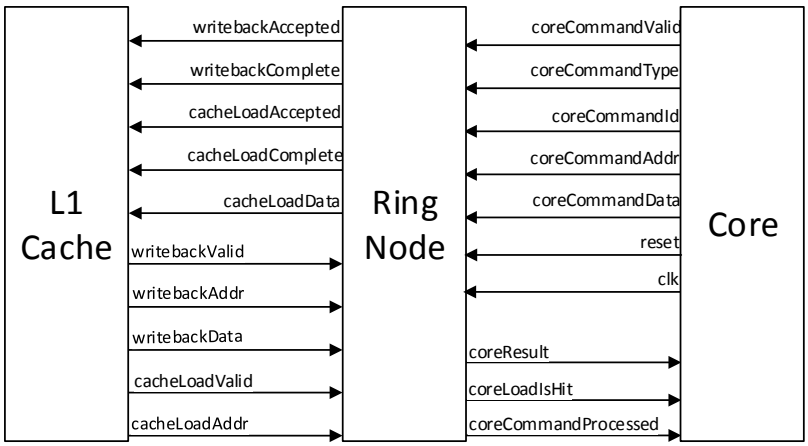



**Figure A.10:** A ring node has direct connections to its local core and its local L1 cache.

are three outputs from the ring node related to the completion of the instruction. A core can only present one instruction at a time to the ring node, and it may not remove it or otherwise present a subsequent instruction unless the ring node indicates completion of the original instruction by raising the one bit *coreCommandProcessed* signal high. All inputs (*coreCommandValid*, *coreCommandType*, *coreCommandId*, *coreCommandAddr*, *coreCommandData*, and *reset*) are assumed to be the outputs of registers – that is, they arrive at the ring node input ports immediately after the rising edge of the clock, and do not change during a clock period. All outputs from the ring node (*coreCommandProcessed*, *coreResult*, and *coreLoadIsHit*) may transition towards the end of the clock cycle, somewhat before the rising edge. The core must be prepared to act on these outputs before the clock edge, by either choosing to hold the instruction for the next cycle, or by preparing to set a new instruction. If *coreCommandProcessed* is raised high, the core must remove the instruction by deasserting *coreCommandValid*, or the ring node will execute the instruction twice. The ring node

outputs are not stable after the clock edge. This dynamic means that the critical path of execution in the ring node may extend into the core logic to set the next instruction. Cores can buffer instructions headed to the ring node, subject to the following constraints:

1. Stores and signals must be presented to the ring node in program order, non-speculatively, as they have global effects in the ring cache.
2. Loads and stores can not be reordered around wait and signal instructions – otherwise, shared data may be read or written outside of the sequential segment.
3. Once an instruction is presented to the ring node (by raising *coreCommandValid* high) it must stay there until released by *coreCommandProcessed*.

### Inputs

coreCommandValid This one bit input should be raised high when the other instruction related inputs from the core to the ring node are valid. It instructions the ring node to execute the presented instruction. This input must remain high until the instruction has completed execution, indicated by *coreCommandProcessed* being set high by the ring node.

coreCommandType This two bit input encodes the four possible intsruction types (wait, signal, load, store). Different instruction types use different sets of the other core inputs, where noted below.

coreCommandId Signal and wait instructions have an associated segment ID. The bit width of this input is dependent on the total number of signals the signal buffer can handle (log2(max_signals)).

coreCommandAddr Load and store instructions set this input to the address to be loaded/stored. In our design, all addresses are 32-bits, and *must* be 4 byte word aligned.

coreCommandData Store instructions set this input to the data to be stored. Wait instructions use the 0th bit to indicate whether the sequential segment they are protecting is empty of shared data accesses or not. This input is 32-bits.

reset This input should be raised high whenever all ring node state needs to be reinitialized – just before a loop invocation starts (or just after a loop invocation ends). Any data stored in the ring cache array should already be flushed to the normal cache hierarchy before this signal is asserted, by executing the special flush signals and waits detailed in Section A.8. During the flush, each ring node writes back only a portion of its stored data, for performance and correctness reasons. So despite the flush, each ring node still needs to invalidate its entire cache array when reset is raised.

### Outputs

coreCommandProcessed This 1 bit signal will be raised high by the ring node when the currently presented instruction from the core is finished executing. For stores and signals, this implies that they have been added to a bundle and are currently being stored to the memory and signal buffer, and beginning propagation around the ring. For loads, this implies that the loaded data is present in *coreResult* and the hit status is set correctly in *coreLoadHit*. For waits, a *coreCommandProcessed* set high indicates that it is safe for the core to enter the sequential segment with the ID that the core set in *coreCommandId*. In all cases, when this signal is set high, the core must prepare to deassert *coreCommandValid* (either that or prepare to set the inputs to another valid instruction) at the next rising edge of the clock.

coreResult This 32-bit output contains the data requested from the executing load instruction.

coreLoadHit In the reference implementation of ring cache, a load will always hit, either by fetching the data from the local ring cache, the local L1, or a remote L1. So from the point of view

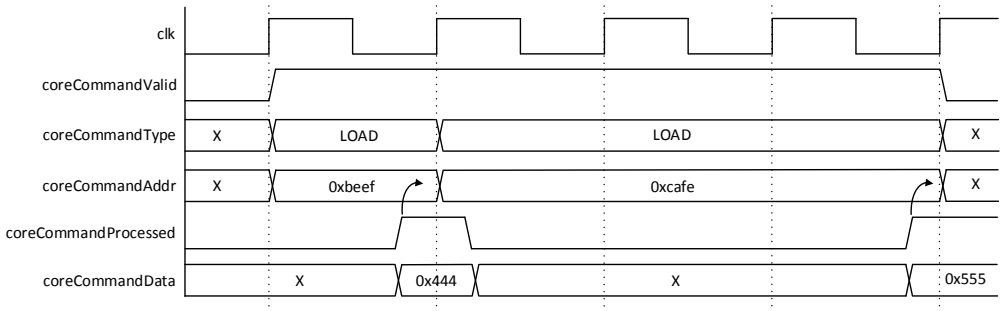



**Figure A.11:** A load that hit in the local ring node completes in one cycle. Another load takes slightly longer.

of the core, every load hits. This 1-bit output could be used to indicate that a load suffered a miss locally, if such information was useful to the core for any reason.

### Instruction Latencies

Depending on current contention for resources, the core may need to hold an instruction at the ring node inputs for several or even hundreds of cycles before it is processed. In the best case, any of the four instruction types can finish executing at the edge of the clock immediately following the inputs being set (i.e., 10 signals could finish executing in 10 cycles, if not blocked for any reason). There are no restrictions on which instructions can follow which – regardless of type, instructions that complete within one cycle can be forever executed back to back (e.g, a sequence of load, store, load, store can finish in 4 cycles if there aren't any resource conflicts or cache array misses).

Loads Loads, if they hit in the local ring node, may return to the core by the next rising edge of the clock. If they miss in the local node, they may need to access the traditional cache hierarchy (perhaps after traversing the request network), so may not return for tens or hundreds of cycles, depending on where they hit in the hierarchy. Figure A.11 shows a timing diagram of a load that finishes in one cycle followed immediately by a load that finishes in 3 cycles. We assume the ring cache memory can give the appearance of combinational reads, either by using an array of registers or a double

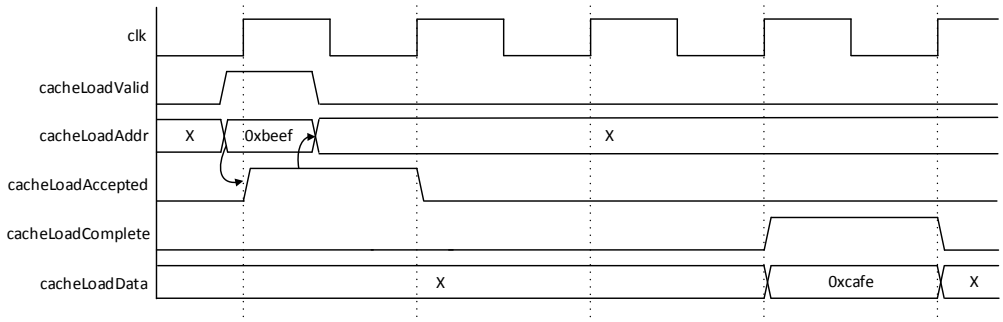



**Figure A.12:** A load from a ring node to its L1 takes a few cycles, as the cache must first accept the load into its internal queues. After processing the load, a *cacheLoadComplete* signal informs the ring node that the loaded data is ready.

clocked SRAM, as will be elaborated on when discussing the memory module in Section A.11.

Stores and Signals Store and signal instructions usually return by the next rising edge of the clock, but may be blocked if the forwarding network is already at capacity or if the memory store port is blocked waiting for a value to be evicted. Generally, stores and signals will not block for more than several cycles.

Waits Wait instructions are only released when enough signals have been received to grant the core entrance into a particular sequential segment. In the case of loops with large loop bodies and low parallelism, this could be many hundreds or thousands of cycles.

### A.6.2 L1 Cache Interface

Every ring node also contains a direct interface to and from its attached core's L1 cache. This interface consists of dedicated wires for both (1) stores resulting from ring cache evictions and (2) loads for requested shared data for which the ring node is the owner. The former input and output wires are prefaced with *writeback* and the latter with *cacheLoad*. Similarly named wires (**Valid*, **Accepted*, **Completed*) have roughly the same semantic for both stores and loads. Like with the core inter-

face, all signals from the cache are assumed to be set immediately after a positive clock edge, and held steady until the next positive clock edge. Signals from the ring node to the cache arrive sometime near the end of the clock cycle, and the cache interface must be prepared to adjust its outputs for the next cycle.

The reference ring cache implementation assumes certain properties about the interface in order to match the cache model in our cycle-level C++ simulator – it may need to be altered if the following guarantees can not be met:

1. All requests from the ring node, once *accepted* by the cache into its internal queues, must be completed in-order. The relative ordering of loads and stores must be maintained.

2. The cache must check for loads or stores to the same address and properly handle read-after-writes. If both a load and store are presented to the cache on the same cycle, the store must always be processed first, even if neither have been *accepted* yet.

3. Once a cache confirms that a store is *completed*, then the store must be visible to the rest of the cache hierarchy in the chip (i.e., the stored value is present in at least the local L1 cache array).

The first two of these restrictions is to ensure that a load to a very recently evicted value doesn't leap ahead of it when accessing the cache, thus potentially reading a stale value from the L1. The ring node does not check that loads or stores to the L1 have the same address, instead relying on the L1 to perform these checks. The ring node currently only implements a single element eviction buffer, so if a load misses in the ring cache, the evicted store is guaranteed to already be presented to the L1 (though not necessarily *accepted* yet). Since a larger eviction buffer would prevent the memory from having to throttle stores, it may be desirable. However, if a larger eviction buffer is used, any loads that miss in the ring cache must make sure to check the eviction buffer before accessing the L1.

The third of these restrictions exists to properly implement the ring cache flush at the end of every loop invocation (see Section A.8). Subsequent code outside of the loop invocation, either between loops or within future loops, may potentially access any memory location from previous loop invocations, so the flushed data must be confirmed to be in the normal cache hierarchy before the ring node can inform the rest of the cores that it is safe to continue.

Figure A.12 depicts a timing diagram for an example load request from a ring node to an L1 cache.

### Outputs

writebackValid, cacheLoadValid These one bit values are set high to indicate that the ring node is either storing or loading a value to/from the L1. They indicate to the L1 that the other outputs from the ring node are valid and may be captured if possible. These signals must remain high until the cache raises the corresponding *writebackAccepted* or *cacheLoadAccepeted* signals to indicate that the store/load has been queued by the cache.

writebackAddr, cacheLoadAddr These 32-bit word aligned addresses corspond to the address that the ring node wants to either store or load to/from the L1. If *writebackValid* or *cacheLoadValid* are raised high, then *writebackAddr* or *cacheLoadAddr* should also be set correctly before the clock edge hits.

writebackData This 32-bit value is the data that the ring node is storing to the L1.

### Inputs

writebackAccepted, cacheLoadAccepted These one bit values are set high by the cache if and only if *writebackValid* or *cacheLoadValid* were set high, and the cache has latched the store/load from the ring node into its own internal queues. They may be raised high as early as the next clock edge after the valid signals are raised high. The ring node, upon seeing an **Accepted* signal raised,

must lower the corresponding **Valid* signal by the subsequent clock edge, or the cache may queue the request twice.

WRITEBACKCOMPLETE This one bit value is set high by the cache when a previously accepted store request is confirmed to be stored in the L1 cache array. This signal is held high for one cycle per item that is confirmed completed. Since many stores may be accepted into internal queues before any have completed, this signal may need to remain high for several cycles in a row as stores are processed, as is often the case during a ring cache memory flush.

CACHELOADCOMPLETE Unlike with writebacks, the ring node only can have one outstanding load to the L1 at a time. When this signal is raised high, the L1 is indicating that the outstanding load has completed, with the loaded data present in *cacheLoadData*. This signal and the loaded data are only valid for one clock cycle.

CACHELOADDATA This is the 32-bit data loaded by the L1 pursuant to a ring node load request. It is only valid only for one clock cycle.

## A.7 Network Interfaces

Every ring node is connected to three unidirectional ring networks. Figure A.13 depicts the network connections from the point of view of a single ring node – incoming links on the left, and outgoing links on the right, connect all the cores in the system into a logical ring. We use the term *bundle* to describe an element which is propagated by the networks. A bundle can be thought of as a network packet with a single flit, though, unlike a traditional network packet, it can contain individual elements from different origins being carried to different final destinations, which all move in lockstep together. Credits, used for flow control, move in the opposite direction of data bundles. Unlike bundles, credits only propagate to the immediate predecessor core, and do not circulate around the

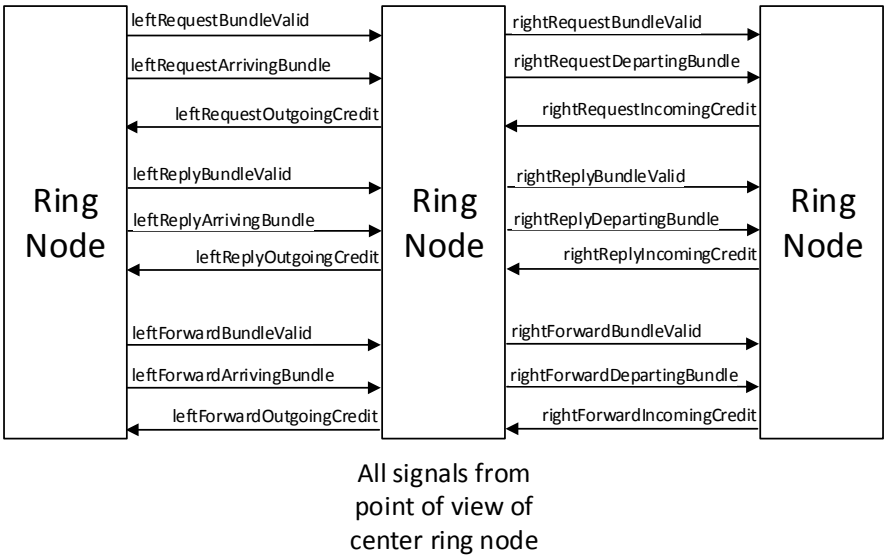



**Figure A.13:** A ring node is connected with its neighbor ring node by three different networks. The forwarding network handles propagation of stores and signals, and the request and reply networks handle loads that need to access a remote ring node or L1 cache. Each network has two arriving (departing) signals for the data payload and a valid bit, as well as a departing (arriving) signal for network credits. All signals are named from the point of view of the center node.

entire ring.

The highest bandwidth and most important of these networks, the forwarding network, is responsible for proactively forwarding stored shared data and signals between every ring node. The details of this network will be described in Section A.9. Two other networks, the request and reply networks, have lower bandwidth and performance characteristics, as they are uncommonly used and only necessary for memory consistency. The reason for and details of these two networks will be described in Section A.10.

The remainder of this section will instead focus on the generic network elements – credit based flow control, buffer sizing, deadlock prevention, and the implementation of the buffer module.

### A.7.1 Credit Based Flow Control

All three of our networks use traditional credit based flow control. Since any of the three networks can stall on a variety of conditions, back-pressure between ring nodes is necessary to avoid overflowing the network receive buffers seen on the left side of Figure A.9. Generally, credit based flow control operates by having each network participant maintain counters for the number of available buffer entries at all possible downstream nodes. In the case of ring cache, each ring node only has one downstream node, so keeps three credit counters total, one for each of the three networks. Anytime a ring node sends a bundle to a downstream node it decrements its count of available downstream buffer entries. Whenever a node removes a bundle from its buffers (either because it has propagated to the subsequent ring node, or has finished circulating around the ring and has been removed from the network entirely), it raises the credit line to its predecessor high, indicating that a buffer entry has become free. The predecessor, seeing the credit line raised, increments its count of available downstream buffer entries. The credit registers can be seen in Figure A.9. The credit counts are incremented when a credit is received, and decremented when a bundle leaves a ring node.

### Buffer Sizing

Credit based flow control potentially introduces an unwelcome latency to bundles propagating in the networks. Consider the case where each node contains only one buffer entry for each network. Further consider that only one ring node, node i, is actively enqueuing items to the forwarding network. After enqueuing one item, it decrements its credit counter, since the downstream node's (node i+1) buffer entries are now full. On the next cycle, the downstream node propagates the item to its next downstream node (i + 2), and sends a credit back to node i. Meanwhile, node i was unable to enqueue an additional item, since it has yet to receive the credit from node i+1. Once it does, some cycles later, it increments its credit counter, and is now free to enqueue an additional item. This delay is known in the network community as the *buffer turnaround time*, and is a direct consequence of the delay in propagating credits back upstream. Generally, a network should have at least enough buffers to prevent stalls of a single enqueuing node, under no other contention. In the case of our ring cache reference design, it takes only one cycle to propagate both data items and credits between cores. As a result, only two buffer entries are needed to allow a single enqueuing core to cycle after cycle send network items without stalling waiting for credits. A larger number of buffer entries could potentially increase performance further, but we did not investigate this possibility.

### Preventing Deadlock

It is important for any network to provide deadlock free routing, since a deadlocked state may be impossible to recover. The ring cache networks only contain single-flit packets, which are simpler to handle than multi-flit packets. However, ring cache still needs to be particularly careful, as the ring structure lends itself to deadlock. We prevent deadlock by enforcing the following, which apply individually to each of the three networks:

1. If there is at least one network buffer entry free in any ring node, a network will always even-

tually be able to make forward progress.

2. A new item (either from the core or from a different network) may not be enqueued into a network if it will consume the last available network buffer entry.

The first of these is guaranteed by the structure of the networks – while the networks can always be blocked on various events (memory evictions, stalls for L1 cache loads, etc), there are no events that can block indefinitely, nor any stall events that have a circular dependence on any other event. The core attached to a ring node isn't involved with dequeuing things from the networks – merely by reaching their final destination, network items will remove themselves without any core intervention. So as long as there is a single buffer entry somewhere in a network, some network item will always eventually be able to move forward one node, and make forward progress. Eventually, even a very highly contended network will drain.

This forward progress is only guaranteed if the second condition is enforced – a new item being added to a network may never take the last available buffer entry. At first this seems like a simple thing to enforce – if there are any items in the receive buffers, don't allow the core to enqueue a new item. Items already in the network must always have priority over newly enqueued items. However, this is not enough. In addition to prioritizing items already in the network, there must be enough buffering at each node, where "enough" is dependent on the link latency between nodes. In the case of our design, where link latency between nodes is only one cycle, we can get away with two buffer entries, as previously mentioned. To understand this, consider the case where every core tries to enqueue items every cycle. After one cycle, every node will have sent a item to its downstream node. Now, since every node has a item in its receive buffer, they will not enqueue any new items to that network. If, during propagation, the network stalls for any reason, items may start to back up behind the stalled item. Nodes downstream of the stalled item may enqueue more items, but as long as their receive buffers are empty, they can do so, knowing that as long as they stop as soon

as they receive a item, they will never take up the last buffer entry in a network. Consider the case, however, where instead of one cycle latency between cores, there is a ten cycle latency between cores, but still with only two buffer entries. All of our ring nodes once again attempt to enqueue as many items as possible. Every node will enqueue two items before stalling on not enough credits. Since the inter-node latency is ten cycles, no node knows if other nodes have enqueued any items yet. After the 10 cycles elapses, every node has two items in their receive buffers, violating our invariant that there must always be a free buffer entry somewhere in the network and creating a deadlock where nothing can make forward progress. To fix this, we need to have at least (inter-node cycle latency + 1) buffer entries per receive buffer. If that were the case in our example, and each node had 11 entries, then each node would only enqueue 10 items before receiving the first one sent from its predecessor. After receiving that first one from its predecessor, a node would stop enqueuing new items. Since there are 11 entries per buffer, this keeps at least one entry free, and our invariant is maintained.

### A.7.2 Buffer Module

Figure A.14 depicts the datapath schematic of a buffer with two entries. Since it is not parameterizable, if more buffer entries are needed (i.e. if link latency needed was higher, more buffer entries are required to prevent deadlock) then it should be redesigned.

#### Parameters

ENTRY_WIDTH This parameter represents the bit width of the stored buffer entries. Depending on the network and the configured signal bandwidth this will change.

#### Inputs

arrivingEntry A receive buffer receives a bundle directly from the network link. It has a bitwidth of *ENTRY_WIDTH*.

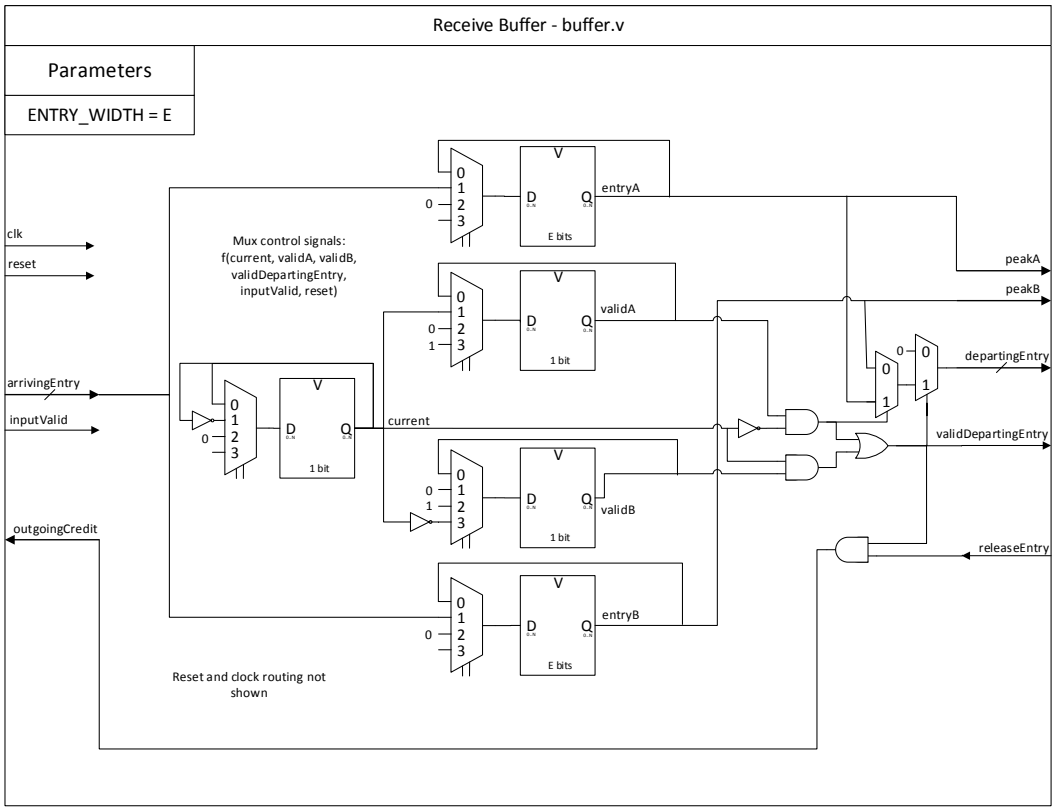
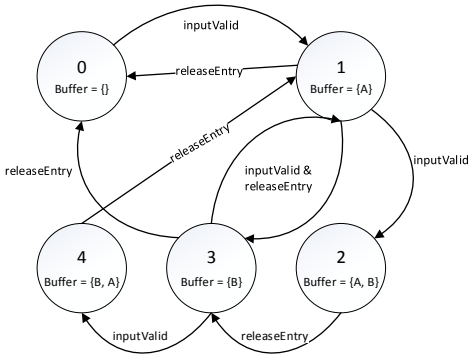



**Figure A.14:** Datapath of the buffer module. Arriving items are stored in one of two buffer entries (A or B). The oldest entry is output to the ring node, which frees the buffer entry by raising *releaseEntry*. Credits are sent upstream to notify the predecessor core that a buffer entry is now available for use.




| State | validA | validB | current | departingEntry | outgoingCredit |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | invalid | 0 |
| 1 | 1 | 0 | 0 | A | releaseEntry |
| 2 | 1 | 1 | 0 | A | releaseEntry |
| 3 | 0 | 1 | 1 | B | releaseEntry |
| 4 | 1 | 1 | 1 | B | releaseEntry |

**Figure A.15:** Control FSM, state bits, and an output of the buffer module. There can either be no valid buffer entries, only A, only B, or both A and B. The current bit tracks which of A or B is the oldest entry, and therefore first to be released. A credit is sent upstream whenever a buffer entry is released.

INPUTVALID This one-bit input is raised high when *arrivingEntry* is a valid network bundle.

RELEASEENTRY This one bit control signal from the ring node informs the receive buffer that the entry at the head of the queue is safe to free, since it is being consumed by the ring node.

## OUTPUTS

DEPARTINGENTRY This *ENTRY_WIDTH* sized output is the entry at the head of the buffer.

VALIDDEPARTINGENTRY This one bit value indicates that *departingEntry* is currently valid.

PEAKA, PEAKB These *ENTRY_WIDTH* sized outputs are the contents of both buffer entries. These are used by the request network to snoop on the forwarding network, to make sure there aren't any loads passing stores to the same address.

OUTGOINGCREDIT This is a one-bit signal sent to the predecessor ring node that is set high when a buffer is freed.

### Datapath

The datapath consists of two registers, one for each entry in our two entry buffer, *entryA* and *entryB*. There are also three other state bits, *validA*, *validB*, and *current*, which indicate which entries are valid, and which is the oldest.

### Control

The control FSM and state bits are depicted in Figure A.15. Control is relatively straightforward. If the buffer is empty, any newly arriving entry is placed in *entryA*. If *entryA* already contains an entry, it is stored in *entryB*. It is impossible, due to the credit flow control, for an entry to be received when both buffers entries are full. At all times, the oldest of the entries is output to the ring node. When *releaseEntry* is raised, the oldest of the two entries is freed, and a credit sent to the predecessor core.

## A.8 Memory Flushing

An important aspect of the HELIX execution model is that there is a memory barrier at the end of every loop invocation, as described in Section A.2. This barrier exists because code outside of a parallel loop may access recently written values from code within the parallel loop. If the core executing the code outside the loop is different from the core that wrote a particular value inside the loop, an incorrect value may be read if there isn't a memory barrier. On a traditional multicore, this barrier can be performed merely by executing the appropriate x86 instruction. On a chip where ring cache is present, however, the contents of every ring node memory need to be flushed to the normal cache hierarchy. This section describes the mechanics of the memory flush at a system level – the individual sections on the signal buffer (Section A.12), the memory module (Section A.11), and the cache interface (Section A.6) touch on the various aspects relevant to those modules and interfaces.

For performance reasons, we desire that the ring cache memory flush happens in parallel, with all

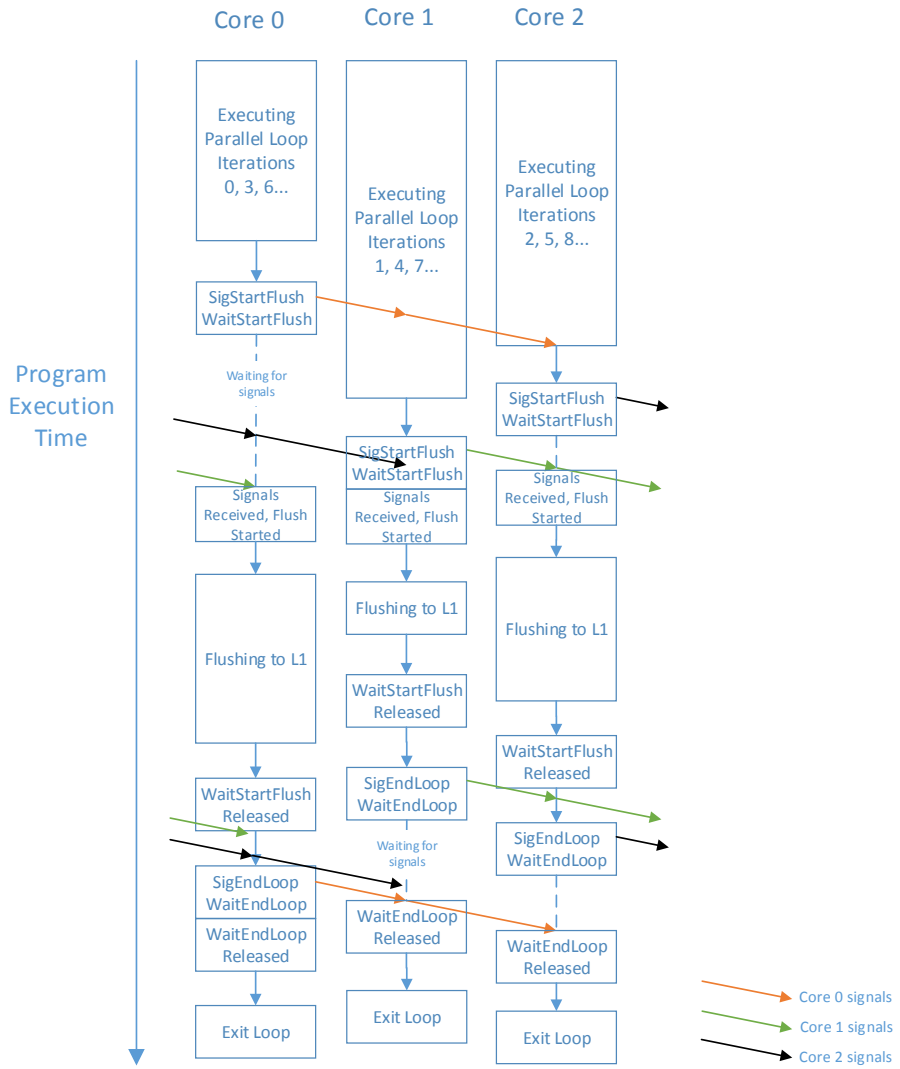



**Figure A.16:** The HELIX required end of loop memory barrier is implemented using two special pairs of wait/signal instructions. First, all cores wait to receive a SigStartFlush from every other core before beginning to flush their local ring node memory. After flushing addresses "owned" by the node to the L1 cache, each core sends a SigEndLoop, and waits for every other core to do the same before leaving the loop.

ring nodes participating at the same time. Later, in Section A.11 we describe the concept of a unique "owner" for every memory address. A core who "owns" a particular memory address, as determined by inspecting the address bits, is responsible for all transfers from/to that address between the ring cache memory and the local L1 cache. During an end of loop flush, all cores can writeback all of their "owned" addresses in parallel. A special bit-array is used so each ring node knows exactly which indexes in their cache array need to be loaded and written back to the L1, to avoid wasting cycles fetching non-owned addresses. On average, each core will writeback approximately (totalNumWordsInArray / numCores) words back to its L1. The memory flush hardware will be described in Section A.11.

At a high level, the flush occurs through the insertion of two special pairs of signal and wait instructions. The function of the first pair is to make each core wait until every other core has finished their last iteration of the executing parallel loop. This forces all stores currently in the forwarding network to finish propagating, so they don't accidentally write to a ring node memory after the flush starts. Since signals can not pass stores in the forwarding network, it is guaranteed that after a core has received the first of these special signals from a particular core, it will not receive any more stores from that core. After a core has received this signal from every other core, it performs its ring node flush. After the flush finishes, a second special signal/wait pair forces every core to wait until every other core has finished flushing before leaving the loop. At that point, it is guaranteed that all data previously stored in the ring cache is now properly stored in the normal cache hierarchy.

Figure A.16 depicts a timeline of a ring cache flush. At the end of every parallel loop, HELIX inserts two pairs of special signal and wait instructions. They *must* be inserted only after the very last instructions of the last iteration executed by a particular core. The highest two sequential segment IDs must be reserved for these two special signals, since the signal buffer tracks them differently from normal signals. Since they are only executed once at the end of the loop, their state in the signal buffer must be initialized appropriately, as discussed in Section A.12.2. Besides using the reserved

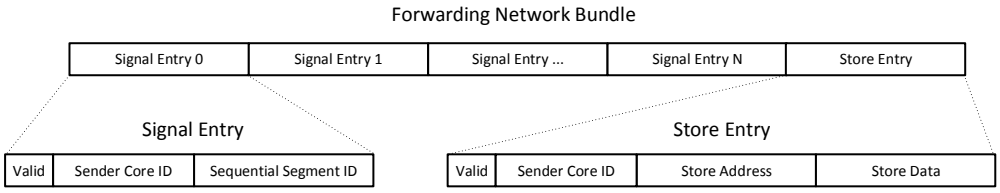



**Figure A.17:** A forwarding network bundle contains one or more entries for signals, and one entry for a store. Both entry types include the ID of the core that sent them to the ring cache, and a valid bit. Signal entries also include a sequential segment ID, whereas store entries include a 32-bit address and data value to be stored. Our reference design uses 128 sequential segment IDs (7 bits to represent each ID), 16 cores (4 bits to represent), and 5 signal slots per bundle. The store entry therefore has 1 + 4 + 32 + 32 = 69 bits, and each signal entry has 1 + 4 + 7 = 12 bits. The total network bundle is 69 + (5 * 12) = 129 bits

sequential segment ID, however, they are encoded just as normal signal instructions. First, every core executes the *SigStartFlush* signal, and then waits on *WaitStartFlush*. Every core must receive the *SigStartFlush* signal from every other core before proceeding. Once a core has received this signal from every other core, it doesn't release the wait instruction quite yet. The special handling of *WaitStartFlush* by the signal buffer, instead of releasing the wait, triggers the ring cache memory flush to commence. After every "owned" address has been confirmed written to the L1 cache, the memory module triggers the release of *WaitStartFlush*. Subsequently, the core may execute a *SigEndloop* and then wait on *WaitEndloop*. Once a core has received a *SigEndloop* from every other core, it knows that every ring node has completed its flush, and that it is safe to leave the parallel loop.

## A.9 Storing Shared Data and Signals - The Forwarding Network

The forwarding network is a key piece of the ring cache. It is responsible for the proactive communication of shared data and signals. In contrast to cache coherence mechanisms which deliver data reactively, only when requested, the forwarding network distributes every store and signal to every other core in the system very soon after they are produced. The forwarding network captures newly executed stores and signals from sequential segments and forwards them throughout the unidirec-

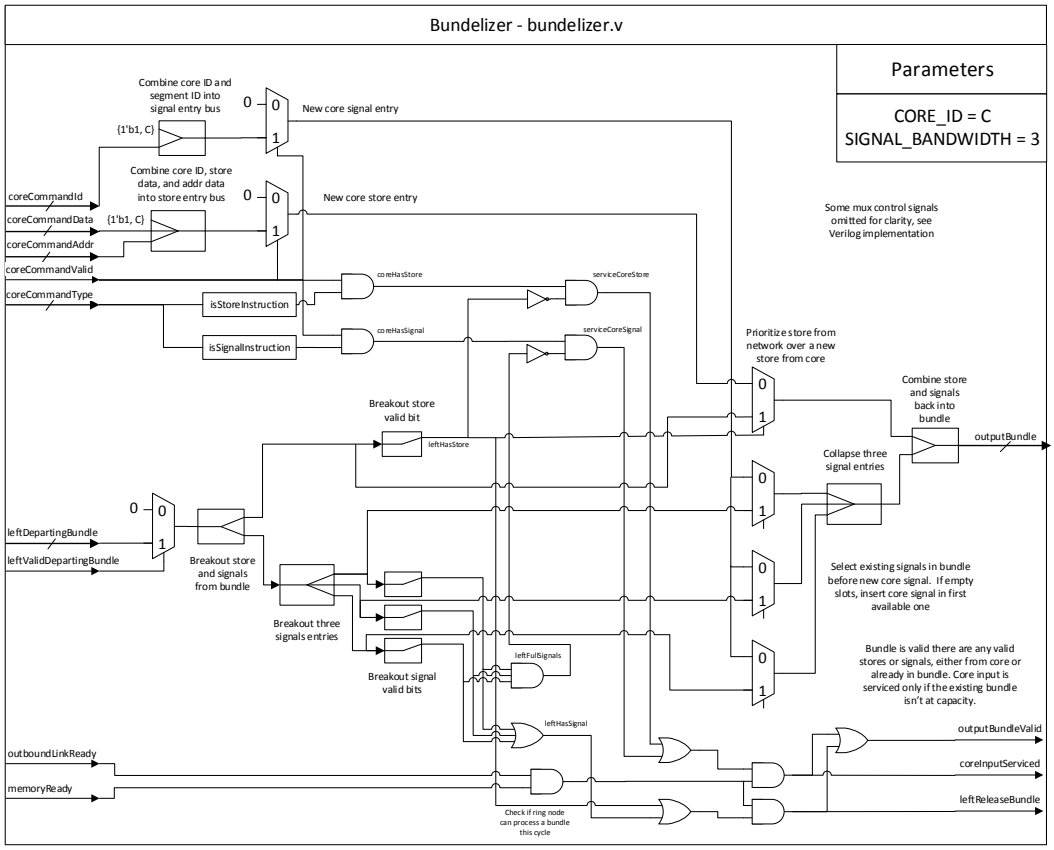



**Figure A.18:** The bundleizer arbitrates between bundles already in the forwarding network, and newly inserted stores and signals from the core. This schematic assumes a signal bandwidth of 3 signals per cycle.

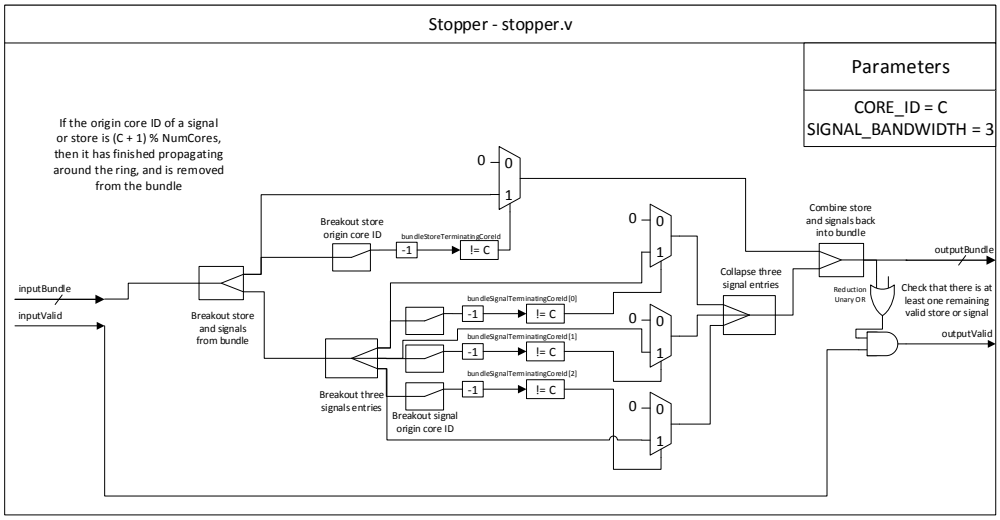



**Figure A.19:** The stopper module removes entries from a bundle if they have finished circulating around the entire ring. It assumes a signal bandwidth of 3 signals per cycle.

tional logical ring of cores that makes up a HELIX-RC system. When a core sends a store or signal to ring cache, it is sent to a module called the bundleizer. This module also reads bundles out of the forwarding network receive buffers (described in Section A.7). If there is room in the network bundle for a new store or signal, the bundleizer grabs it from the core. This module passes output to the rest of the ring node only if the ring node memory is ready to accept a store instruction, and there are available downstream receive buffers in the subsequent ring node. The output is sent to three places. First, if there is a valid store in the bundle, it is sent to the ring node memory to be stored. Second, any signals in the bundle are sent to the signal buffer to be recorded. Third, the entire bundle is sent to the stopper module. The stopper module is responsible for removing any stores or signals from the bundle that have circulated around the entire ring (e.g., a signal inserted by core 0 will be removed from the bundle by the stopper module of core 15, after passing through core 1, 2, etc.). If there are any stores or signals remaining in the bundle after the stopper has pruned it, the remainder of the clock cycle is used to send the bundle over the forwarding network link to the sub-

sequent core in the ring. In this way, the stores/signals are written to the memory / signal buffer in parallel with network link transmission – once the bundleizer outputs a bundle, it is guaranteed access to the memory, signal buffer, and network link in parallel.

In the remainder of this section, we will first describe what a bundle is. Then, we will present schematics for the bundleizer and stopper modules.

### A.9.1 Network Bundle

The atomic unit that is circulated around the forwarding network is called a *bundle*. Figure A.17 depicts a bundle, which consists of one or more signal entries and a single store entry. When a core presents a store or signal to the ring node, the bundleizer adds it to one of the available signal or store slots. Since ring cache can only process one store per cycle, it doesn't make sense to provide more than one store slot. The signal buffer, on the other hand, can process multiple signals per cycle. Higher signal bandwidth improves performance for a couple of the SPEC CINT 2000 benchmarks, so it is desirable to package multiple signals in a bundle.

Once added to a bundle, individual stores and signals all move in lockstep together – it is impossible for an individual store or signal to "move ahead" a bundle or be "left behind" in a bundle. This is a requirement of the HELIX execution model – if signals were able to pass stores, they may allow a core entry into a sequential segment before the shared data they are protecting has arrived. For this reason, propagation of the entire bundle must stall if the ring node memory is not ready to process a new store (e.g., if its currently evicting a value to the L1).

### A.9.2 Bundleizer Module

The bundleizer module is entirely combinational. It performs its duties sometime early in the clock cycle. This module is responsible for:

1. Adding new signals and stores from the local core to an existing bundle in the network if

there is an empty slot of the appropriate type.

2. Preventing a core from enqueuing a new signal or store into the network if there isn't a slot.

3. Creating a new bundle for a signal or store from the local core if there aren't any bundles currently in the network receive buffer.

4. Deciding if a bundle on this cycle can be sent to the memory, signal buffer, and propagated to the subsequent ring node.

The first and second of these items upholds the requirements of preventing network deadlock in Section A.7.1. By always deferring to bundles already in the network, a newly inserted store or signal from the core is unable to consume the last available buffer entry in the network.

### Parameters

CORE_ID The bundleizer module must know the local core ID of its attached core. The ID gets added to any newly initialized store or signal entry when the local core injects a new signal or store into the ring node.

SIGNAL_BANDWIDTH The number of signal slots in the network bundle must be known. The bundleizer needs to know which signal slots are invalid so it can add a new signal from the core to the bundle.

### Inputs

All core inputs to the ring node (*coreCommandId*, *coreCommandData*, *coreCommandAddr*, *coreCommandValid*, *coreCommandType*) related to a new instruction are forwarded to the bundleizer. See Section A.6 for a description of these signals. In short, they represent all of the information necessary to enqueue a new store or signal from the core to a bundle.

leftDepartingBundle This input carries the oldest bundle currently in the forwarding network receive buffers. This bundle only leaves the receive buffer if instructed by the bundleizer, by default it is only presented to the bundleizer for inspection. The size of this input depends on the configured signal bandwidth and how many bits are required to represent a sequential segment ID, as shown in Figure A.17.

leftValidDepartingBundle This one bit input is held high if the *leftDepartingBundle* input is valid.

outboundLinkReady This input from the ring node is held high if the number of forwarding network credits is greater than 0, indicating that the ring node downstream in the network has at least one buffer entry available to be used.

memoryReady This one bit input is held high if the local ring cache memory can accept a new store instruction this cycle.

### Outputs

outputBundle This output is a full network bundle. The only difference from the *leftDepartingBundle* input is that one new store or signal from the core may have been added if there was space. It is the same bitwidth as *leftDepartingBundle*.

outputBundleValid This one bit output merely indicates whether the *outputBundle* signal is valid.

coreInputServiced This one bit output is raised high if and only if a a new store/signal presented by the core was accepted into a network bundle. By raising it, the bundleizer is letting the core know that the store or signal is finished executing, since it will be written this cycle. As a result

the core is free to execute another instruction.

leftReleaseBundle This one bit output is raised high if the bundleizer processed the bundle at the head of the receive buffer (contained within *leftDepartingBundle*). Upon raising it, the receive buffer knows it should release the buffer entry and send a credit to the upstream ring node.

### Datapath

Figure A.18 depicts a schematic of the bundleizer module, with a *signal bandwidth* of 3 signals per cycle. If both *outboundLinkReady* and *memoryReady* are high, then the bundleizer can potentially send a bundle to the *outputBundle* output port. If there is already a bundle in the network, in *leftDepartingBundle*, the bundleizer inspects it to determine whether there are any empty slots to insert a new signal or store from the core inputs. If there aren't any core inputs, the bundle is simply passed to the output port. If there is an available store slot (and the core is trying to insert a store), the core ID of the ring node is appended to the *coreCommandData* and *coreCommandAddress* and the result is added to the bundle, which is then passed to the output port. If there are any available signal slots (and the core is trying to insert a signal), the first available slot in the bundle is taken. The mux control signals not pictured on the schematic are set by the valid bits of the signal entries in the bundle, which determine whether there is a slot available for the new core signal, and if so, which slot it should take. If there isn't a valid bundle already in the receive buffer, then a new bundle is constructed with only the new core store or signal. The bundleizer indicates to the core with the *coreInputServiced* output if the store or signal was accepted. It indicates to the receive buffer with the *leftReleaseBundle* signal whether the input bundle was passed on to the output, and therefore the entry can be released from the receive buffer.

### Control

The bundleizer is stateless. All control signals are determined from the valid bits of the input bundle or *coreCommandValid*.

## A.9.3 Stopper Module

The stopper module removes any stores or signals from the network bundle that have completed circulating around the ring, while allowing stores and signals that haven't completed circulating to continue over the forwarding network link to the subsequent core. Essentially, it checks the origin core ID of each store and signal slot, and invalidates the entry in the outgoing bundle if it is ready to be removed.

### Parameters

CORE_ID The stopper module must know the local core ID of its attached core, so it can check which of the items in the bundle have completed circulation around the ring.

SIGNAL_BANDWIDTH The number of signal slots in the network bundle must be known, so the stopper module knows how many signal slots it needs to check.

### Inputs

inputBundle The input bundle is the output bundle from the bundleizer.

inputValid This one bit input is high if the *inputBundle* signal contains any valid store or signal entries.

### Outputs

outputBundle This bundle is the same as the *inputBundle*, with any store or signal entries invalidated if their origin core ID belongs to the subsequent core in the ring.

outputValid This one bit input is high if the *outputBundle* signal contains any valid store or signal entries.

### Datapath

This module merely inspects the origin core IDs for each signal and store entry in the bundle. If the origin core ID of an entry minus 1, mod the number of cores equals the core ID of this node, the stopper zeros out all of the bits in that entry. Whatever signal or store was there ceases to exist in the forwarding network. If all slots in the bundle were invalidated this cycle, then *outputValid* is set low, and the entire also bundle ceases to exist. The schematic is shown in Figure A.19.

## A.10 Loading Shared Data - The Request/Reply Networks

One of the most important contributions of ring cache is that it transforms a *reactive* cache coherence protocol into a *proactive* system of communication. In most cases, when a core attempts to access shared data, it will find that it is present in its local ring node memory. The communication cost is only the time it takes to access the ring node. If the normal cache hierarchy was used, it would take dozens of cycles from the time data was requested until the data arrived locally

One of the distinguishing factors of ring cache over other fast communication mechanisms (Multiscalar register file [54], scalar operand networks [59]) is that the number of shared pieces of data is not known at compile time, nor the number of consumers of any particular shared piece of data. Since other solutions rely on a statically known number of shared elements, they are not suitable for HELIX. Instead, a cache structure that can handle an unknown number of elements is needed. That

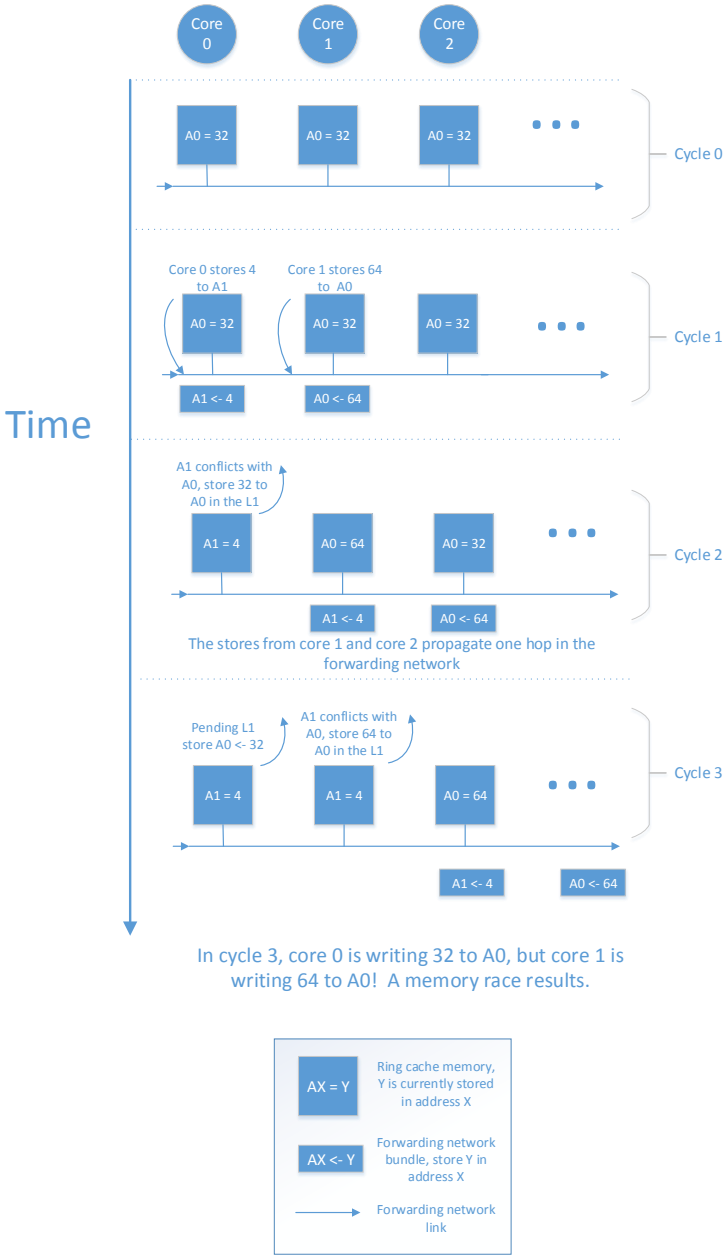



**Figure A.20:** Allowing any core to writeback and load to the normal cache hierarchy results in a race condition that may violate correctness!

means ring cache must be able to handle load misses and cache evictions, since there is no guarantee that all of the shared data will fit in the ring cache. For this reason, the normal cache hierarchy must be relied upon to support the ring cache.

However, the memory consistency guarantees of the normal cache hierarchy are different than that of ring cache. A naive integration between the two raises a significant consistency issue. Consider an implementation where a ring node simply writes-back any evicted value to its local L1. In the case of a subsequent ring cache load miss, the ring node fetches the data from its L1. While this might seem like a reasonable idea at first glance, it gives rise to race conditions which will violate correctness. Figure A.20 depicts the timeline of a three-core system suffers from such a race condition. For simplicity we assume a ring node memory size of just a single word. Sometime in the past, the value of 32 is written to address A0, which is propagated on the forwarding network and stored in every ring node memory. In cycle 1, core 0 and core 1 execute two different sequential segments. Core 0 stores the value of 4 to address A1, and core 1 stores the value of 64 to address A0. Both stores enter the forwarding network. On the next cycle, core 0's store has triggered an eviction in its memory, and A0, with a value of 32, begins to be written back to core 0's L1 cache. In the same cycle, core 1's ring node memory updates address A0 with the value 64. Additionally, both stores propagate one more hop on the forwarding network. On cycle 3, the store to A1 triggers an eviction of A0 in core 1. Now, core 1 begins to writeback A0 to its own L1. Unlike with core 0, however, core 1 writes-back the newly updated value of 64. This results in a race to update A0 with either the old or new value. Many (or perhaps all) modern architectures do not make any guarantees about which value will be recorded first.

We handle this consistency issue by enforcing the constraint that for any unique memory address, there is a single *owner core* that is solely responsible for any loads or stores to that address between the ring cache and the normal cache hierarchy. The owner core is determined based on the address – details of how this is done can be found in Section A.11. In the case of a ring cache load miss, the re-

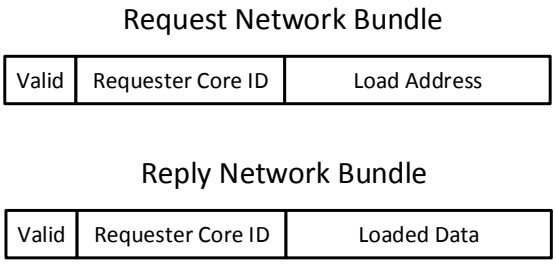



**Figure A.21:** Unlike the forwarding network, the request and reply networks only contain one load (request or result) per bundle. The core ID indicates which ring node requested the remote load, which is where the loaded data will be returned to. For a 16 core system, with 32-bit addresses and data, the total number of bits per request or reply bundle is 1 + 4 + 32 = 37 bits.

quest network is responsible for requesting the data from the owner ring node, which subsequently performs a L1 lookup, and returns the result on the reply network. In the case of a ring cache eviction, if the ring node owns the evicted address, it writes it back to its L1. If the evicting ring node is not the owner, it simply discards the data without performing any writeback.

The load unit module of ring cache handles all load-related interactions between the core, the request network, and the reply network. Loads from the core enter the load unit, which arbitrates between the core and the request network for access to the memory module. We first discuss the request and reply networks at a high level before describing how the core and the two networks interact within the load unit.

### A.10.1 Request and Reply Networks

The request and reply networks, as mentioned, exist only to fulfill memory consistency requirements. They implement a reactive data transfer mechanism that, if used frequently, completely eliminates any of the benefits of having the ring cache. For this reason they are not tuned for performance – if they aren't used rarely, then performance will tank regardless. Like the forwarding network, the request/reply networks also implement unidirectional ring networks that single hop around each core in the system, one clock cycle per hop. While it might seem like a higher connec-

tivity topology could have been used, since there isn't a requirement for strict in-order data flow of loads like there is for stores and signals, there are two reasons why we stuck with unidirectional rings. First, unidirectional rings are easy to reason about in terms of data flow and deadlock avoidance. Second, and more importantly, care must be taken such that a remote load on the request network doesn't pass a store to the same address on the forwarding network, or an incorrect value could be returned. Both of these reasons will be discussed in the following subsections.

Unlike the forwarding network, which contains multiple signals (and a store) per network bundle, the request and reply networks contain only a single remote load request/reply. Figure A.21 depicts the structure of each network bundle – for the request network, the core ID of the requesting core is included, along with the address to be loaded. In the reply bundle, the requesting core ID remains, but the address to be loaded is replaced with the actual loaded data. If the load unit decides a ring node can not not service a load locally (i.e. it is not the owner of the address), it adds its core ID and the address to a request network bundle, and injects it into the network. At every hop around the ring, the local ring node load unit inspects the request network bundle to see if owns that particular memory address. If so, it removes it from the request network, and first performs the load operation on its ring node memory. If it misses in the ring node memory, the value is subsequently loaded from its L1. The loaded data is then packaged in a reply network bundle and injected into the reply network. At each hop around the reply network, each ring node inspects the core ID in the bundle to see if the reply is destined for it. Once it reaches the core that originally performed the load, it is removed from the reply network, and the result handed to the core. This behavior results in a remote load latency of at least 16 cycles just for network transmission – since the rings are unidirectional, the total number of hops to the owner and back to the original core will always equal the total number of cores. There are additional cycles of latency for entering/exiting the network, performing the load to the ring node memory, potentially accessing the L1, etc.

### Reducing Remote Loads

The number of remote loads can be optimized be relaxing our constraint that only the *owner core* of an address can interact with the normal cache hierarchy. Instead, only the owner core of an address can interact with the normal cache hierarchy *if the address has previously been written to ring cache, but subsequently evicted*. The race condition previously shown in Figure A.20 can only occur if an address was at some point present in the ring cache – in that example, A0. If a particular core is trying to load A0, but it knows for sure that A0 was *never* written to the ring cache yet this loop invocation, it can conclude that it is impossible that it is currently being evicted from any other node. Given the definition of a sequential segment, if a core is loading A0, no other core in the system may be loading or storing it. It is therefore safe for the core to load it from the normal cache hierarchy, even if it isn't the owner. We use a bloom filter in each ring node to track whether an address has been written yet this loop invocation. If a core attempts to perform a load, and it misses in its ring node memory, the bloom filter is consulted. If the address is present in the bloom filter, the core knows it must make a remote load request unless it is the owner of the address. If it is not present in the bloom filter, the core can load from its own L1, despite not being the owner. More details of the bloom filter can be found in Section A.11.

### Deadlock Avoidance

Our general approach for deadlock was previously explained in Section A.7.1. As mentioned, one of the reasons for choosing unidirectional rings for the request/reply networks was because we could apply the same policy that the forwarding network implements to avoid deadlock. The basic approach is to make sure a new item being injected to the network can never take the last available buffer entry anywhere in that network. That way, forward progress is always guaranteed. In our current reference design, each core can only make one remote load request at a time – the number of

outstanding remote loads, then, is equal to the total number of cores in the system. Since each network buffer contains two entries, they can never fill up, in either the request or reply network. Since a possible optimization is to allow multiple outstanding remote loads at a time, we also enforce the invariant that a new item being injected into either network has lower priority than items already in the network. As long as there are sufficient buffer entries considering the link latency (again, as detailed in Section A.7.1), this prevents deadlock even if cores can make multiple remote requests at a time.

In the case of the request network, the only source of new items is when a core injects a remote load. Items exit this network when they arrive at the owner core of the requested address. There, they wait until they can access the local ring node memory, perform the load, and then wait to enter the reply network. Until they can enter the reply network, backpressure is applied on any other items needing to exit the request network, creating a unidirectional dependence between the request network and reply network. Once in the reply network, the item propagates until it reaches the original requesting core, where it exits the network. As long as the reply network makes forward progress, as guaranteed by keeping at least one buffer entry empty at all times, the request network will also be able to make forward progress. Since the reply network doesn't depend on the request network for progress, there is not a circular dependence, so deadlock is avoided.

### Ordering Constraints

Unlike the forwarding network, there aren't as many ordering constraints on the request/reply networks. The forwarding network enforces a strict ordering between signals and stores, since a signal passing a store can produce incorrect behavior. Loads in the request/reply networks don't have a similar constraint – the loads could hypothetically be reordered, though we don't do that. The one very important constraint, however, is that a load in the request network must never pass a store in the forwarding network to the same address. While exceptionally rare (it doesn't occur a single time

in any phase of any of our benchmarks), it is hypothetically possible. Imagine the case where core 0 is within a sequential segment, and performs a store to address X (owned by, let's say, core 8), which begins propagating in the forwarding network. Immediately after, it performs a store to address Y, which conflicts in the local ring node memory, so evicts address X. Right after that, core 0 attempts to load address X. Since address X was previously stored in the ring cache, but was just evicted, and core 0 is not the owner, it must make a remote load request. Imagine that the store to address X is stalled from forwarding at core 2 due to the eviction of an unrelated address. The remote load request by core 0 could hypothetically pass the stalled store to address X at core 2, and proceed to core 3, 4, and eventually 8. At core 8, the load will either load a stale value from the local ring node memory, or a stale value from the L1. Either way, by passing the store in the forwarding network, an incorrect value is loaded. For this reason, a ring node prevents the request network from propagating if *any* of the buffer entries in its forwarding network contain a store to the same address that is being remotely loaded. In our example, the remote load in the request network would stall at core 2 until the forwarding network continued propagating. Once the store to address X was processed by core 8 (i.e., left the forwarding network receive buffer), the remote load would be allowed to access core 8's ring node memory. The reply network, on the other hand, does not need to perform any checks of this nature, since it only delivers already loaded values back to the requester.

### A.10.2 Load Unit Module

The load unit module is responsible for arbitrating between loads from the request network and the core, which both want to access the local ring node memory. It is responsible for sending the load to the actual ring node memory module. Additionally, it contains the logic to both inject and remove items from the request and reply network. It also returns loaded values from the ring node memory or reply network to the core. In some ways, it serves a similar purpose that the bundleizer module does for the forwarding network.

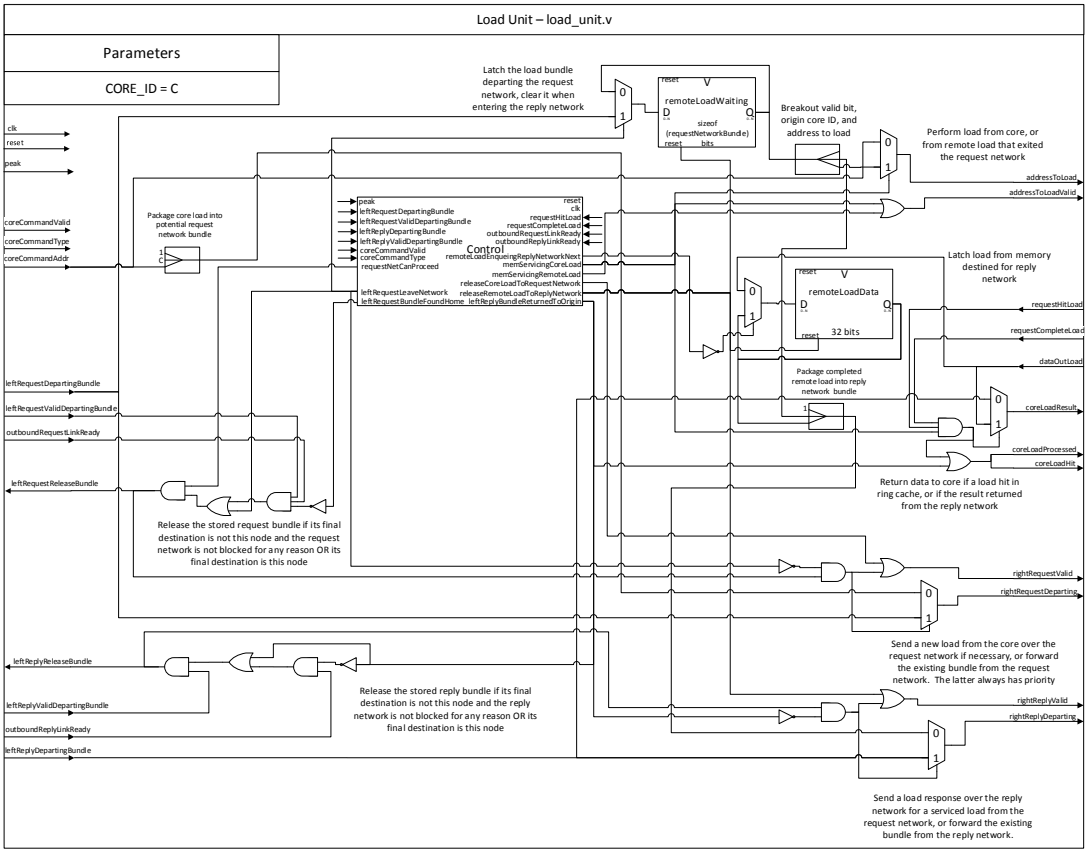



**Figure A.22:** A simplified schematic of the load unit module datapath.

Figure A.22 depicts a datapath schematic for the load unit, which includes connections to/from the core, from the request and reply network receive buffers, and to the outgoing request and reply network links. For simplicity the control logic is not shown. The control FSM and Verilog should be consulted for control behavior.

### Parameters

CORE_ID The load unit needs to know the ring node's ID, so it can 1) append it to the remote loads it adds to the request network and 2) detect when a remote load in the request network needs to exit the network to access its ring node memory.

### Inputs

All core inputs to the ring node *coreCommandAddr*, *coreCommandValid*, *coreCommandType* related to a load instruction are forwarded to the load unit. See Section A.6 for a description of these signals. In short, they represent all of the information necessary to enqueue a new load instruction to the ring node.

reset The one-bit reset signal shouldn't be necessary, as all outstanding state should be zeroed appropriately when remote loads complete execution.

peakA, peakB These signals carry the addresses from the entries currently in the forwarding network receive buffers. They are inspected to make sure loads in the request network don't pass stores to the same address.

leftRequestDepartingBundle, leftReplyDepartingBundle These inputs carry the oldest bundle currently in the request or reply network receive buffers, respectively. These bundles only leave the receive buffer if instructed by the load unit. The structure of these signals is shown in

Figure A.21.

leftRequestValidDepartingBundle, leftReplyValidDepartingBundle These one bit inputs are held high if the corresponding *leftXDepartingBundle* input is valid.

outboundRequestLinkReady, outboundReplyLinkReady These inputs from the ring node are held high if the number of request/reply network credits is greater than 0, indicating that the ring node downstream has at least one buffer entry available for the corresponding network.

requestCompleteLoad This one bit signal from the memory module indicates that the load sent to it has completed executing, with the resulting data (in the case of a hit) and hit status stored to *dataOutLoad* and *requestHitLoad*. A load may complete on the same cycle as it was sent to the memory, as our reference design implements the memory with a combinational register array.

requestHitLoad This one bit signal from the memory module is high if the recently completed load either hit in the local ring node memory, or was properly loaded from the node's L1 interface (if it either was the address owner, or the address was not present in the bloom filter).

dataOutLoad This 32-bit signal contains the loaded data from a load sent to the memory module.

### Outputs

leftRequestReleaseBundle This one bit output is raised high if the load unit is either 1) forwarding the bundle from *leftRequestDepartingBundle* to the outgoing request network link or 2) removing the remote load from the request network to process locally. In both cases, the buffer entry from the request network receive buffer is consequently invalidated.

leftReplyReleaseBundle This one bit output is raised high if the load unit is either 1) forwarding the bundle from *leftReplyDepartingBundle* to the outgoing reply network link or 2) removing the reply from the reply network, since this node is the one who initiated the remote load to begin with. In both cases, the buffer slot from the reply network receive buffer is consequently invalidated.

rightRequestDeparting, rightReplyDeparting These signals correspond to the outgoing network links of the request and reply networks. They are each sized according to the network bundles in Figure A.21.

rightRequestValid, rightReplyValid These one-bit outputs are raised high if the load unit is sending a network bundle on the respective network links, *rightRequestDeparting* and *rightReplyDeparting*.

addressToLoad This 32-bit signal holds the address of the load that the load unit is sending to the memory module.

addressToLoadValid This 1-bit signal is high if *addressToLoad* is a valid address that should be loaded.

coreLoadResult This 32-bit signal is the loaded data value that the load unit is returning to the core. The data may have originated from the local memory module, or from the reply network following a remote load.

coreLoadHit This 1-bit signal is always high, since from the point of view of the core (in the current implementation), loads always hit, just with a variable latency.

coreLoadProcessed This 1-bit signal is raised high when *coreLoadResult* and *coreLoadHit* are valid, and the load instruction the core is waiting on has been fetched from either the local memory module or from the request/reply networks.

### Datapath

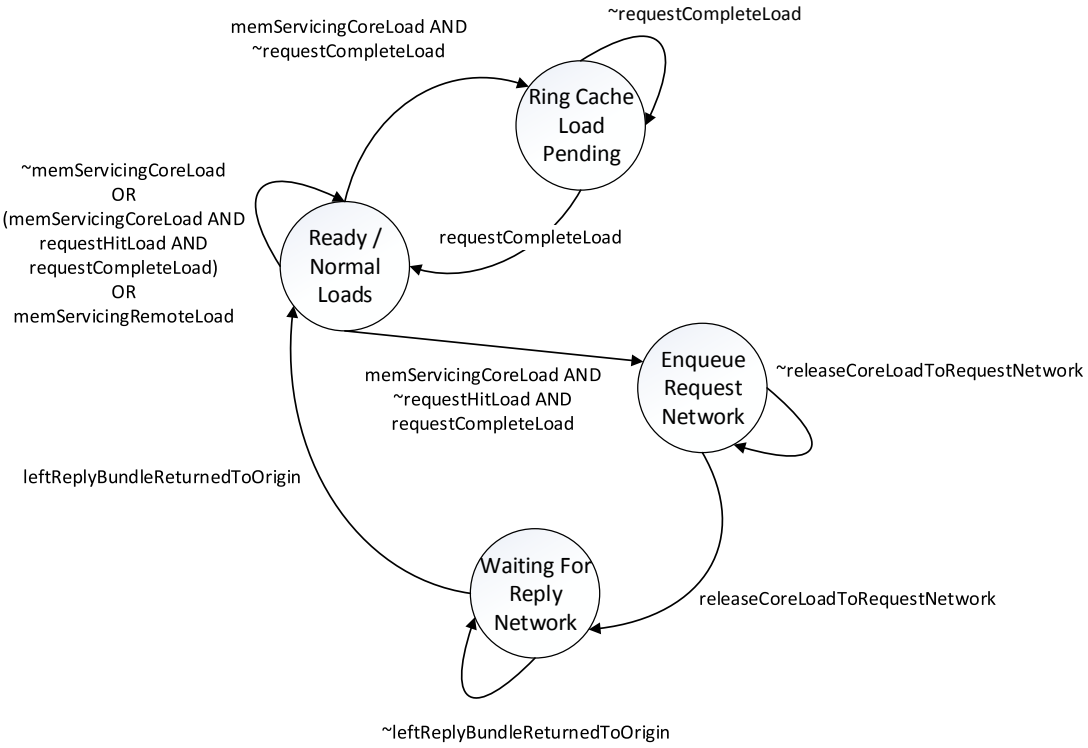
Loads from core Loads directly from the core interface arrive near the beginning of a clock cycle on *coreCommandValid, coreCommandAddr* and *coreCommandType*. The load unit makes sure the core's new command is a load. If it is, and if there isn't currently a load from the request network accessing or waiting to access the memory module (that is, *memServicingRemoteLoad* is low), the core's requested load address is sent to the *addressToLoad* and *addressToLoadValid* outputs. The memory module attempts to load the address from the local ring node memory. If it hits, or if it loads from its L1, the result is returned to the load unit in *dataOutLoad*, with a high value on *requestHitLoad* and *requestCompleteLoad*. The result is routed back to the core on *coreLoadResult*, with *coreCommandProcesed* raised high to let the core know the load has finished. If it misses locally and is unable to access the L1, *requestCompleteLoad* goes high, but *requestHitLoad* stays low. This triggers a request network bundle being constructed with the *coreCommandAddr*. The core waits until the request network link is ready and the local request network receive buffer is empty, and then sends the network bundle over the link, on *rightDepartingBundle*. This request travels to the adjacent node by the end of the cycle. It continues throughout the network until it arrives at the owner node of the address, where it accesses the L1, fetches the requested value, and returns it on the reply network. On every cycle, the output of the reply network receive buffer, *leftReplyDepartingBundle*, is inspected. If the core ID in the reply network bundle matches the local core ID, the bundle is removed from the reply network, and its data payload returned to the core on *coreLoadResult*, with a high value on *coreCommandProcessed*. The core is now free to send a new instruction to the ring node.

Loads from request network The load unit also processes loads from the request network. On every cycle, the requested address in *leftRequestDepartingBundle* is inspected to see if it is owned by the local core. If it is, it is removed from the request network, and the whole bundle is latched in *remoteLoadWaiting*. Once the memory module is available (i.e., *memServicingCoreLoad* is low), the load is sent to the memory via *addressToLoad*. The result, either from the ring node memory or from the L1, is eventually returned on *coreLoadResult*, which is latched in *remoteLoadData*. There, the data waits until the reply network link is ready and the local reply network receive buffer is empty. Once this is true, a reply network bundle with the fetched data from *remoteLoadData*, and the core ID from *remoteLoadWaiting* are packaged into a bundle and sent over the *rightReplyDeparting* network link, where they travel to the subsequent core by the end of the cycle. Meanwhile, *remoteLoadWaiting* and *remoteLoadData* are cleared. Eventually, the reply bundle returns back to the original requesting core.

Request/reply network forwarding On every cycle, the load unit also checks if the bundles at the front of the request and reply buffers, in *leftRequestDepartingBundle* and *leftReplyDepartingBundle*, need to continue propagating to the next ring node. If their corresponding *outboundXLinkReady* inputs are high, and they are not supposed to exit the network at this node, they are simply forwarded to the corresponding output links, *rightRequestDeparting* or *rightReplyDeparting*. The request network bundle also checks the forwarding receive buffer output (the *peak* inputs) to make sure its remote load address doesn't match any of the store addresses being forwarded. It stalls if any addresses match.

### Control

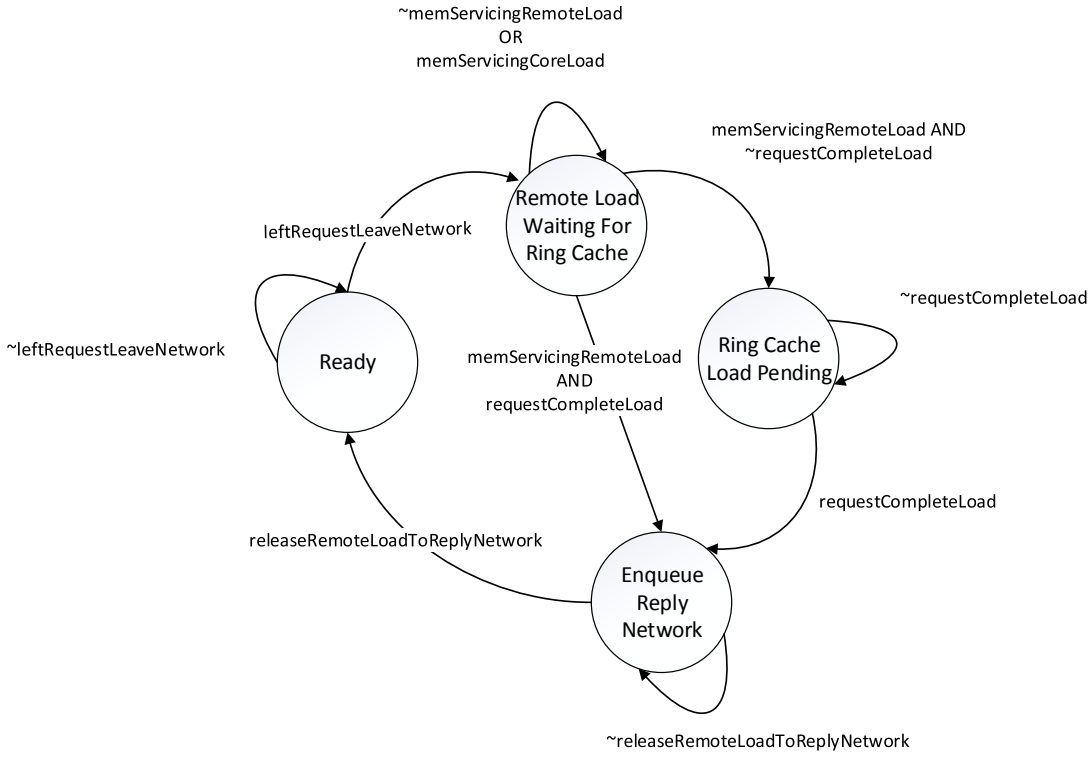
The control FSM is split into one for core loads, in Figure A.23, and one for remote loads, in Figure A.24. The FSMs are only partially independent – since the memory module can only handle




memServicingCoreLoad and memServcingRemoteLoad are mutually exclusive, since the ring cache can only process one load at a time

| State | pendingCoreLoad | coreLoadEnqueuingRequestNetwork | coreLoadWaitingForReply |
|---|---|---|---|
| Ready | 0 | 0 | 0 |
| Ring Cache Load Pending | 1 | 0 | 0 |
| Enqueue Request Network | 0 | 1 | 0 |
| Waiting For Reply Network | 0 | 0 | 1 |

**Figure A.23:** The FSM and state bits for core loads to the load unit.




memServicingCoreLoad and memServcingRemoteLoad are mutually exclusive, since the ring cache can only process one load at a time

Unlike core loads, remote loads always register as hits, because even if they miss in the ring cache, they will always access the attached L1 of the remote ring node.

| State | pendingRemoteLoad | remoteLoadWaiting | remoteLoadEnqueuingReplyNetwork |
|---|---|---|---|
| Ready | 0 | 0 | 0 |
| Remote Load Waiting For Ring Cache | 0 | 1 | 0 |
| Ring Cache Load Pending | 1 | 0 | 0 |
| Enqueue Reply Network | 0 | 0 | 1 |

**Figure A.24:** The FSM and state bits for request network loads to the load unit.

one load at a time, the core or the request network may need to wait if one or the other is already accessing it. Additionally, the request network load (in *remoteLoadWaiting*) always his priority over a new core load, since it blocks all items behind it in the request network, and its requester has already suffered significant latency waiting for it. If the load unit is merely forwarding bundles over the request and reply networks, rather than injecting or removing bundles, then these FSMs do not apply. Consult the Verilog for exact control signal logic.

### Loads from core

Ready/Normal Loads In this state, the core may initiate a new load if the memory is not busy servicing another load. If the load hits in the local ring node memory, as indicated by *requestHitLoad* and *requestCompleteLoad* going high by the end of the cycle, then it returns the loaded result to the core, and stays in the Ready state. If the load doesn't return immediately, because the L1 is being accessed (either because the core is the address owner, or the address is not in the bloom filter), then it transitions to the Ring Cache Load Pending state. If the load returns immediately, but with *requestHitLoad* indicating that it missed in the local ring node memory and couldn't be fetched from L1, it transitions to the Enqueue Request Network state.

Ring Cache Load Pending This state waits for a core load to return from the memory module. If it is in this state, then the load is accessing the local L1. Once it eventually returns as a hit, it transitions back to the Ready state.

Enqueue Request Network In this state, a core load has already attempted to access the local ring node memory, but the load missed, and was unable to be fetched from L1. Here, the FSM waits until it is safe to enqueue the remote load into the request network. For deadlock avoidance, *releaseCoreLoadToRequestNetwork* goes high only if the request network link is ready (i.e., has at

least one credit), and there isn't an item currently in the request network receive buffer (i.e., *leftRequestValidDepartingBundle* is low). Even if *leftRequestValidDepartingBundle* is low, however, *releaseCoreLoadToRequestNetwork* can also go high if the item in the request network receive buffer is exiting the request network to the load unit this cycle. Once the remote load is injected onto the request network links, the FSM transitions to the Waiting For Reply Network state.

Waiting for Reply Network The core remains in this state until its remote load has been serviced by a remote core, and the loaded data returned over the reply network. This could be a potentially very long time, depending on network contention, and whether the load hit in the remote ring node's memory, or if it had to go to its L1. The incoming reply network link is inspected every cycle, and when the core ID in the reply network bundle matches the local core ID, the data is returned to the core, and the FSM transitions back to the Ready state.

### Loads from request network

Ready In this state, the load unit waits until there is a bundle in the request network receive buffer whose load address is owned by the local ring node. Once there is, it is removed from the request network and stored in the *remoteLoadWaiting* register. The FSM transitions into the Remote Load Waiting For Ring Cache state.

Remote Load Waiting For Ring Cache In this state, the latched remote load waits until the ring node memory is free. Like with loads from the core, the load address is sent to the memory. If the data returns immediately (*requestCompleteLoad* goes high before the end of the cycle), then the load hit in the local ring node memory. The data is stored in *remoteLoadData*, and the FSM transitions to the Enqueue Reply Network state. If the load doesn't return by the end of the cycle, it must have missed in the local ring node memory, and is accessing the L1. Unlike with core loads,

these remote loads that miss in the ring node memory will always access the L1, since they were removed from the network specifically because they arrived at the address owner node. In this case, the FSM transitions to the Ring Cache Load Pending state.

Ring Cache Load Pending This state waits for a remote load to return from the memory module. If it is in this state, then the load is accessing the local L1. Once it eventually returns as a hit, it transitions to the Enqueue Reply Network state.

Enqueue Reply Network In this state, the load from the request network has obtained its value, either from the ring node memory or the local L1. It waits until it is safe to enter the reply network. It is safe (as far as deadlock avoidance is concerned) to do so when the reply network link is ready (i.e., has at least 1 credit) and the reply network receive buffers are empty (i.e., *leftReplyValidDepartingBundle* is low). It is also safe to do so if the item at the front of the reply network receive buffer is exiting the network this cycle. A reply network bundle is constructed with the loaded data as a payload, and is sent over the reply network links to the subsequent core. The FSM returns to the Ready state.

Request/reply network forwarding As previously mentioned, the request and reply networks can propagate any bundles already in the network as long as the outgoing network links are ready (i.e., have at least one credit.) Additionally, the request network can only propagate if the address of its remote load doesn't match any addresses being stored in any of the bundles in the local forwarding network buffers. For deadlock avoidance, propagating existing network bundles always has priority over injecting new items.

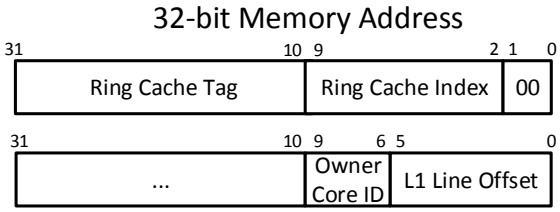



**Figure A.25:** For a 1 KB direct-mapped cache with a line size of a single 32-bit word, there are 8 index bits. For 16 cores, 4 bits of the address are required to determine which core owns it. Since owners must be assigned on L1 cache line granularity, the bits above the L1 line offset are used to determine ownership. For a typical 64 byte L1 line size, the ownership bits completely overlap with the ring cache array index bits, which results in simpler hardware to track which indexes a particular ring node needs to flush.

## A.11 Ring Cache Memory

The memory module of ring cache facilitates fast access to the shared data contained within. It caches data written by cores within sequential segments, and provides the cores fast access to the same data when it is read within sequential segments. The memory has two logical ports – one dedicated for loads, and one dedicated for stores. For optimal performance, the actual storage within the array module is currently implemented with a register array, which enables a combintional load, a combinational tag read, and an edge triggered write to occur in a single cycle. While it could be made to be set-associative (as in the original HELIX-RC paper), our reference implementation uses a direct-mapped structure (performance was only very slightly impacted for one benchmark), which makes certain other optimizations more straightforward. The cache line is a single word, as dictated by the compiler to avoid false sharing. Each ring node memory module also interacts with the rest of the cache hierarchy for any memory addresses that it is the owner of (for load misses and evictions). A bloom filter is used to increase the number of load misses that a ring node memory can fetch from its local L1, rather than requesting from a remote core's L1. At the end of every loop invocation, the memory is responsible for flushing its contents back to the L1, as mentioned in Section A.8.

Before describing the contents of the memory module in detail, we first describe the concept of

an *owner core*, in addition to the different read and write modes of the ring cache memory. Then we go on to describe the memory module and its submodules, the array module and the bloom filter module.

### Address Ownership

For correct execution, access to shared data within sequential segments must be sequentially consistent. This property is guaranteed by the compiler's placement of *wait* and *signal* instructions. These instructions, in concert with the ring cache hardware, guarantee that cores can only read or write shared values in correct loop iteration order, within the appropriate sequential segments.

The ring cache memory size is limited, so at some point it must interact with the normal cache hierarchy. The core may try to load addresses that were previously evicted or haven't been written to the ring cache yet – perhaps those written by a previous parallel loop, or by code outside any loop. Additionally, conflicts in the ring cache memory will result in evicted values needing to be written back to the L1. The latter presents a problem, since it in effect means that any ring node could write any particular address to the L1 at any point in time, even outside of a sequential segment. This is even more concerning since the address being evicted may also be currently circulating throughout the forwarding network. If this is the case, two cores (one preceding the circulating store, and one following it), may writeback two different values for the same address, and a race ensues. This scenario is described more in depth in Section A.10.

For this reason, we establish the notion of unique *core ownership* of a memory address. Figure A.25 depicts how the *owner core* is derived from a 32-bit address, in addition to the ring cache array index and tag, for our reference design of 1 KB of storage, an assumed L1 line size of 64 bytes, and 16 cores. In order to avoid false sharing of L1 cache lines, which would incur cache coherence communication costs, different cache lines are assigned different owner cores. Owner cores are selected by using the log2(numCores) bits immediately above the L1 line offset. Notice that the owner

core ID of an address completely overlaps with the ring cache index bits. This implies that, for our reference configuration, certain ring cache array indexes always map to the same owner core. This coincidence makes it easier to tell which values a ring node will need to writeback to the L1 at flush time. Since there are only 256 sets in 1 KB of memory, only 16 bits per core are required (256 / 16 cores) to keep track of which of these indexes currently holds an owned address. If the owner core ID didn't exclusively map to ring cache indexes this way, 256 bits would be required per core.

### Reads

Reads have two modes of operation. In normal mode, incoming addresses access the ring cache memory array. Since the array is implemented with registers instead of SRAM, the result of the load is known combinationally, before the end of the cycle. If the load was a hit, the result is returned to the requester, the load unit module. However, if the load misses in the ring cache, and the attached core is the owner of the address, then the memory module fetches it from the L1 instead. It then returns the result to the load unit, indicating a hit. If the load misses in the ring cache, but the attached core is *not* the owner, the load unit is informed that the load missed, and subsequently initiates a remote request through the request/reply networks to another core's L1 to fetch the value.

Many of these remote requests may be avoided if its known for sure that the address has *never* been written to the ring cache yet during this loop. If we know that this is the case, it is impossible that the address is currently being written back by some other remote ring node. It is therefore safe in this case for a ring node that is not the owner of the address to load it directly from its L1. By adding all written addresses to the bloom filter module, a load that misses in the ring cache memory is able to determine whether it can just access the local L1, instead of suffering a costly trip around the request/reply networks. Since bloom filters only have false positives, and no false negatives, a poorly sized bloom filter can result in extra remote requests, but never incorrect behavior.

### Writes

Writes have three modes of operation. For typical stores, incoming addresses perform the tag lookup combinationally, and the write itself at the clock edge. Simultaneously, the *ownerBitset* is updated to reflect whether the array index it was written to now contains an owned address. The address is also inserted into the bloom filter. If the array index that was written previously contained a valid value, and the attached core is the owner of its address, it is written to the local L1 interface. If the core is not the owner, the value is simply discarded. The memory module doesn't allow any more writes (which stalls the forwarding network) until the L1 has confirmed the value has been written to the normal cache hierarchy. Having only one outstanding eviction simplifies the logic, and allows the ring cache to rely on the L1 cache to satisfy any loads to recently evicted values, as detailed in Section A.6.2. If the eviction buffer is made larger, than any loads that miss in the ring cache memory must search it to make sure they aren't bypassing stores to the address they are loading from.

At the end of every loop invocation, the signal buffer instructs the memory module to flush its contents to the L1. The memory module inspects its *ownerBitset* to determine which ring cache array indexes contain owned addresses, and writes them back to the L1 one by one, clearing the *ownerBitset* as it goes. After the L1 confirms that every flushed value has been written to the normal cache hierarchy, the memory module informs the core that the flush has completed.

## A.11.1 Memory Module

Figure A.26 depicts a simplified schematic of the memory module. It contains the bloom filter and array submodules. The reference implementation uses a 1 KB direct-mapped cache, since performance is not noticeably less than when using a set-associative cache. For simplicity, certain control logic is not shown, and some control signals were left unrouted. For fully commented control signals and output logic, please see the Verilog implementation of the module.

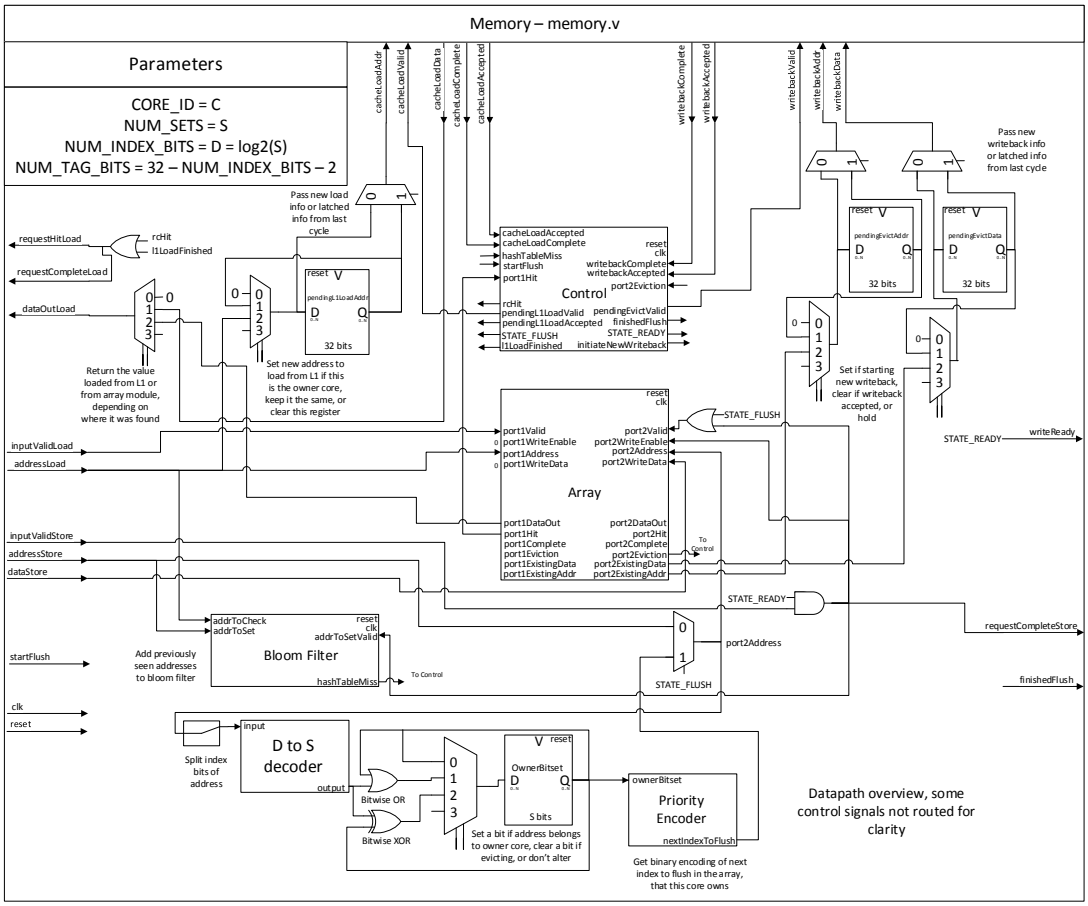



**Figure A.26:** A simplified schematic of the memory module datapath.

### Parameters

CORE_ID The memory module must be aware of the core ID so it can determine if it is the owner of an address.

NUM_SETS In our reference design we use 256 sets. With a line size of a single 4 byte word, this makes the total cache size 1 KB.

NUM_INDEX_BITS, NUM_TAG_BITS An address is split into index and tag bits, as shown previously in Figure A.25.

### Inputs

reset The reset signal is used to clear all state after the end of a loop invocation. In particular, the memory array and bloom filter bits must be cleared for proper operation.

addressLoad This 32-bit address input comes from the load unit module, and it is the load to be performed this cycle. It is stable early in the clock cycle.

inputValidLoad If high, *addressLoad* is valid, and the load should be performed.

addressStore, dataStore A 32-bit address in which to store the 32-bit data value. Originates from the output of the bundleizer module. Should be stable early in the clock cycle.

inputValidStore If high, *dataStore* should be written to *addressStore* this cycle.

startFlush This one-bit output from the signal buffer module is raised high when it has received a flush signal from every other core. It causes the memory to transition from normal operation into the flush state.

writebackAccepted, writebackComplete These 1 bit signals related to writebacks to the L1 are directly connected to the cache interface. See Section A.6.2 for details.

cacheLoadAccepted, cacheLoadComplete These 1 bit signals are related to loads to the L1 cache. See Section A.6.2 for details.

cacheLoadData This 32-bit signal is the data returned by a load to the L1. See Section A.6.2 for details and timing.

### Outputs

writeReady This one-bit signal is raised high if the memory module can accept a store this cycle (i.e. it is not stalled performing a writeback). Since it is based on state bits, it is stable at the beginning of a clock cycle. The bundleizer module uses this signal to know if a circulating bundle in the forwarding network can enqueue its store and continue propagating. The forwarding network does not need a write completion confirmation from the memory to continue propagating, only the ready signal to initiate a new store.

requestCompleteLoad This one-bit output indicates to the load unit that a load has completed. In the case of an immediate ring cache hit, it may be raised as soon as the end of the clock cycle that *inputValidLoad* was raised high. It may be delayed several or tens of cycles if a miss in the ring cache resulted in an L1 load.

requestHitLoad This one-bit signal is raised high if the load hits in the ring cache, or if the memory module loaded the value from the L1. It is only held low (indicating a miss) if the load can not be satisfied by this ring node, and instead must be sent over the request/reply networks. This is only the case if the ring node is not the address owner, and the address has been previously written to the ring cache (i.e., the address is present in the bloom filter). This signal is valid only when

*requestCompleteLoad* is raised.

dataOutLoad This 32-bit signal contains the loaded data, and is valid only in the case that *requestHitLoad* is high.

writebackAddr, writebackData These 32-bit address and data wires are the values to be written to the L1 cache. See Section A.6.2.

writebackValid A one-bit signal that is raised high if the memory module is performing a writeback to L1. See Section A.6.2.

cacheLoadAddr A 32-bit address that the memory module wants to load from the L1. See Section A.6.2.

cacheLoadValid This one-bit signal is raised high if *cacheLoadAddr* is a valid load to be performed. See Section A.6.2.

### Datapath

Loads Loads enter the memory module through the *addressLoad* and *inputValidLoad* inputs near the beginning of a cycle, originating from the load unit. There, they access a dedicated port of the memory array to load the data. Note that port1 of the memory array has the write signals sent to 0 – in our current implementation the port is dedicated to reads, but this doesn't necessarily need to be the case. In parallel with the array load, the bloom filter is checked to see if this memory address has been previously written to the ring cache. If the load hits in the ring cache memory, the resulting data is returned on the *dataOutLoad* output, with *requestHitLoad* and *requestCompleteLoad* being set high. In this case these signals are valid at the end of the same clock cycle the load began, since our memory array performs combinational reads. If the load misses in the ring cache, it either

is requested from the L1 cache interface, or returned as a miss with *requestHitLoad* set low and *requestCompleteLoad* set high, which indicates to the load unit that the load needs to be sent out over the request/reply networks. The load can be serviced by the local L1 in two conditions – first, if the owner of the address to be loaded is the local core, or if the *hashTableMiss* output of the bloom filter is high, indicating that this address has not been written to the ring cache previously. If the load is sent to the L1, the *cacheLoadAddr* and *cacheLoadValid* outputs are set appropriately. Once the cache responds, potentially many cycles later, *requestHitLoad* and *requestCompleteLoad* are set high, and the loaded data is passed on to *dataOutLoad*.

STORES Stores are sent to the memory module by the bundleizer module on the *addressStore*, *dataStore*, and *inputValidStore* inputs. These signals are routed to port2 of the memory array where they perform a tag lookup combinationally, and write the new value from *dataStore* into the array on the clock edge. In parallel, *addressStore* is routed to the *addrToSet* input of the bloom filter, where it is hashed and added to the filter bitset. Also in parallel, the index of the address is used to write the corresponding bit in the *ownerBitset*. If the address is owned by this core, the bit whose position matches the array index of the address is set to 1, otherwise to 0. In our reference design, since the bits of the address used to determine the owner core overlap completely with the cache array index bits, and we use a direct-mapped cache, only a limited subset of bits in the *ownerBitset* could ever be set – the rest are optimized away by synthesis.

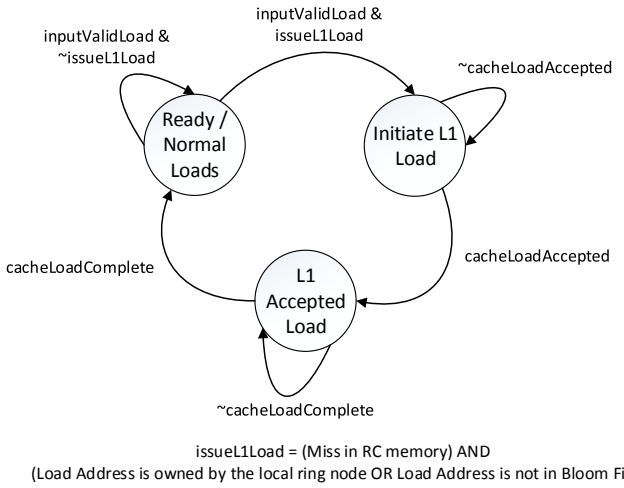
If the store to the memory array triggered an eviction, the evicted address/value may need to be written to the L1 cache. If this ring node is not the owner of the evicted address, no action is taken. If it is the owner, the address/value is sent to the L1 cache writeback interface. While a value is being written back, the *writeReady* output is held low, to prevent any more stores from being performed until the L1 cache has confirmed the write, which could take several cycles or more. After completing the writeback, *writeReady* returns high to allow a new store to be performed.

Flush When the memory module receives a high value on the *startFlush* input, it transitions to the flush state. In this state, the *ownerBitset* is consulted to determine which indexes of the cache array must be written back to the L1. The *ownerBitset* array is fed into a priority encoder, which gives a binary representation of the least significant bit set in the bitset. This represents an index of the cache array that must be written back to the L1, since it is owned by this core. The memory module fetches this data value from the array, and initiates a writeback to the L1, like for an eviction. Unlike for normal evictions, the flush mechanism doesn't wait for the writeback to complete, only that it is accepted by the cache interface. Meanwhile, the *ownerBitset* clears the bit corresponding to the index that was just written back, and uses the output of the priority encoder to fetch the next array index to flush. Once the *ownerBitset* is equal to 0, *finishedFlush* is raised to indicate to the rest of the ring node that the flush is done.

### Control

Figure A.27 depicts the FSM for any loads to the memory module, whereas Figure A.28 depicts the FSM for stores, writebacks, and flushes. For loads, the control states are fairly simple. In the *Ready/Normal* state, loads that hit in the ring cache are performed at the rate of one cycle per load. If a load misses in the local ring cache, and it is either owned by this core or is not present in the bloom filter, an L1 load begins by setting the *cacheLoadValid* and *cacheLoadAddr* outputs. If neither condition is true, the load immediately returns as a miss. Once the load is accepted by the cache interface, we transition to another state where we wait for the cache to return the loaded value. Once it does, the loaded data is returned and normal loads resume.

A similar dynamic happens for evictions from the ring cache, except with the writeback interface instead of the load interface. For stores, the *Ready/Normal* state processes one store per cycle as long as there aren't any evicted values to writeback. If there are, the store port transitions to initiate the L1 writeback, where the evicted address and value are presented to the cache interface. Once they are

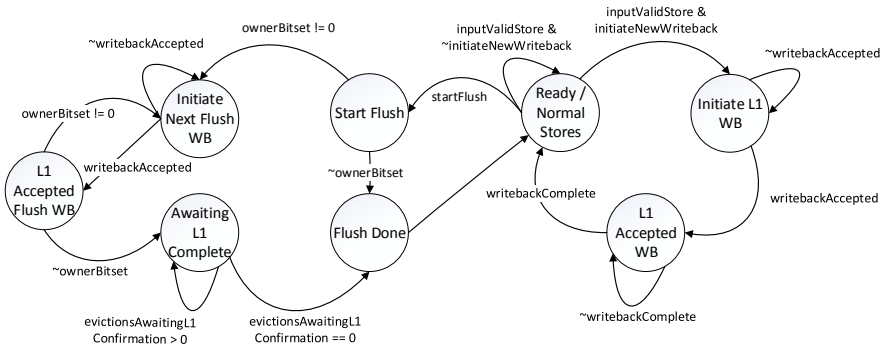



| State | readState | pendingL1LoadValid | pendingL1LoadAccepted |
|---|---|---|---|
| Ready | `STATE_READY | 0 | 0 |
| Initiate L1 | `STATE_L1_LOAD | 1 | 0 |
| L1 Accepted | `STATE_L1_LOAD | 1 | 1 |

**Figure A.27:** The FSM and state bits for memory module loads.

accepted by the interface, we transition to a waiting state, where we remain until the cache confirms the store has completed.

For flushes, the logic is slightly more complex. When the *startFlush* signal is received, the port transitions to the *start flush* state. If there aren't any bits set in the *ownerBitset*, then the node has nothing to flush, so transitions back to the *Ready* state. If there are bits in the *ownerBitset*, then the corresponding array indexes are first fetched, then written to the cache interface, while the bit is cleared in the *ownerBitset*. Unlike with normal evictions, we don't wait until stores are confirmed, only that they are accepted by the cache. A count of the number of writes that have yet to be confirmed are saved in the *evictionsAwaitingL1Confirmation* register, which is incremented for every initiated writeback, and decremented whenever *cacheLoadComplete* goes high for a cycle. After




evictionsAwaitingL1Confirmation increments every time a new flushed item is accepted by the L1 cache, and decrements whenever the L1 confirms that it has completed a store

| State | writeState | pendingEvictValid | flushWalkDone | evictionsAwaitingL1 Confirmation |
|---|---|---|---|---|
| Ready | `STATE_READY | 0 | 0 | 0 |
| Start Flush | `STATE_FLUSH | 0 | 0 | 0 |
| Initiate Next Flush WB | `STATE_FLUSH | 1 | 0 | Previous value + 1 - cacheLoadComplete |
| L1 Accepted Flush WB | `STATE_FLUSH | 0 | 0 | Previous value - cacheLoadComplete |
| Awaiting L1 Complete | `STATE_FLUSH | 0 | 1 | Previous value - cacheLoadComplete |
| Flush Done | `STATE_FLUSH | 0 | 1 | 0 |

initiateNewWriteback =
(A normal store triggers an eviction AND
the evicted address is owned by the local ring node)
OR
(Flush is active and there are still more data items to writeback)

| State | writeState | pendingEvictValid |
|---|---|---|
| Ready | `STATE_READY | 0 |
| Initiate L1 WB | `STATE_PENDING_EVICTION | 1 |
| L1 Accepted WB | `STATE_PENDING_EVICTION | 0 |

**Figure A.28:** The FSM and state bits for memory module stores and flushes.

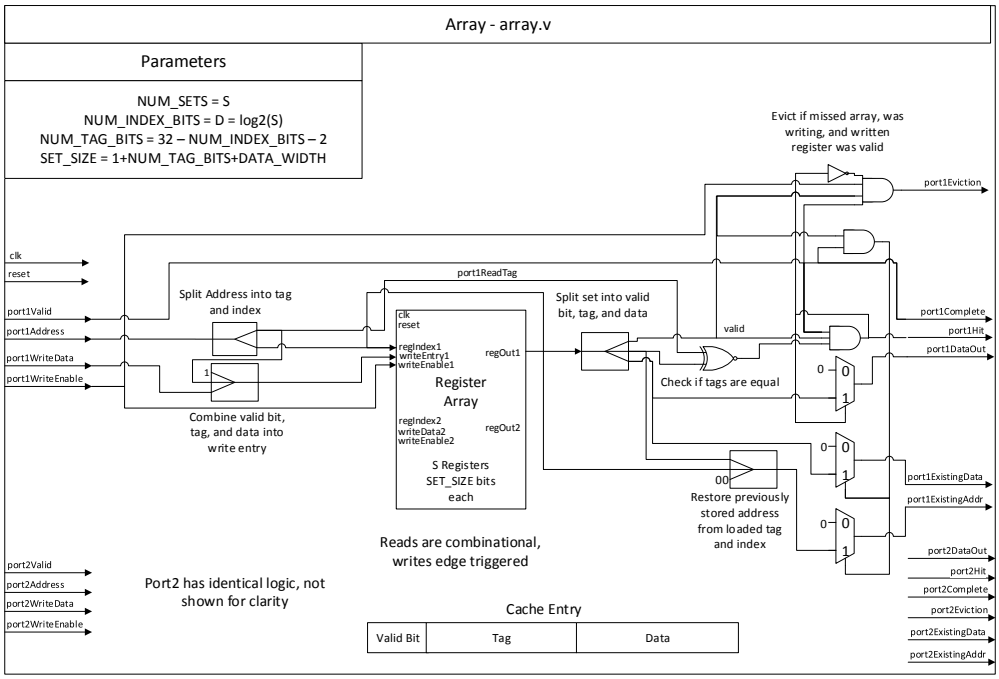



**Figure A.29:** A schematic of the array submodule. The tag and data storage of ring cache is implemented with an array of registers to enable combinational loads and tag lookups.

*ownerBitset* is completely empty, we idle until *evictionsAwaitingL1Confirmation* has reduced to 0, and then transition back to normal operation.

### A.11.2 Array Module

Figure A.29 depicts the array submodule. It provides a two read/write port interface to an array of registers. This array of registers serves as the primary storage for the ring cache memory. Each register in the array contains a valid bit, a cache tag, and a data value. Since ideally we want ring cache to be able to process a load and a store simultaneously in one cycle, the combinational reads provided by the register array allow for a tag lookup and store in the same cycle, as well as a load that does not require a clock edge. If it is found that this performance is not needed, an SRAM could be substituted in this module. Some control logic in the array module and memory module will need to be

tweaked to account for loads requiring a clock edge, and stores requiring two cycles.

The array module treats both port1 and port2 equally, even though port1 is dedicated for loads, and port2 is dedicated for stores and the flush. Synthesis will remove any unnecessary extra logic.

### Parameters

NUM_SETS  This is the number of sets in the register array / cache. Since the current implementation just uses a direct-mapped structure, the total cache capacity is the number of sets multiplied by the size of a word, 32-bits.

NUM_INDEX_BITS  The number of index bits needed to address *NUM_SETS* cache entries.

NUM_TAG_BITS  The number of tag bits required to differentiate addresses in the cache.

SET_SIZE  The number of bits in a cache entry is 1 (for a valid bit) plus *NUM_TAG_BITS* plus 32 bits for data.

### Inputs

Since port1 and port2 of the array have identical logic, descriptions will provided generically.

reset  The one-bit reset signal must be raised between loop invocations to clear out every register in the array.

portXValid  This one-bit signal indicates whether the other inputs are valid.

portXAddress  A 32-bit address of to be loaded or stored to/from the array.

portXWriteEnable  If this is low, perform a load. Otherwise, perform a tag lookup and a store.

portXWriteData A 32-bit data value to be stored in the array. Only valid if *portXWriteEnable* is high.

### Outputs

Since port1 and port2 of the array have identical logic, descriptions will provided generically.

portXComplete This one-bit signal is raised high if the outputs of the module are valid. In our current implementation, since the array can be read combinationally, all accesses are complete by the end of the cycle that they become valid inputs.

portXHit This one-bit output is raised high if an access to the array resulted in a tag match.

portXDataOut In the case of a hit, this 32-bit output contains the accessed data from the array.

portXExistingData This 32-bit output contains the data that was contained in a cache entry before a write occurred. It is used to writeback a value that has been evicted, or during a memory flush.

portXExistingAddr This 32-bit output contains the address that was previously stored to a cache entry before a write occurred. Along with *portXExistingData*, it is used to to writeback an evicted/flushed piece of data to the L1 cache.

portXEviction This one-bit signal is raised high if a store missed in the array, but the index it accessed previously contained valid data, which must now be written back to L1.

### Datapath

The array module contains typical cache logic. First, the address on the *portXAddress* input port is split into index and tag bits, depending on the size of the register array. The index bits are used

to access one of the registers in the array, which contains the appropriate cache entry. Meanwhile, if *portXWriteEnable* is high, the data to be written from *portXWriteData* is packaged into a cache entry by combining a valid bit, tag bits from the address, and the data itself. Combintionally, the proper register in the array is read, either in service of a load or a tag lookup for a store. The resulting read entry is split into a valid bit, tag, and data. If the valid bit is not set, then either a load has missed, or a store has nothing to evict. If the valid bit is set and the tag bits of the requested address and the read cache entry match, then either the load or store has hit. If the request was a store, no eviction is required. The loaded data is placed on *portXDataOut* and *portXHit* is set appropriately. If the tag bits don't match, then either a load has missed, or in the case of a store, the previously stored value must be evicted. The previously stored value will be found on *portXExistingData*, along with its reconstructed address on *portXExistingAddr*. At the clock edge, if *portXWriteEnable* is high, then the constructed write entry is stored to the register indicated by the input address' index bits.

#### Control

Other than writing the appropriate register on the clock edge, there isn't any control to speak of.

### A.11.3 Bloom Filter Module

Bloom filters consist of a bit array and one or more hash functions. They are used as a resource efficient way to track membership of a set, but without the cost of storing every member of a set. Items to be inserted into the bloom filter are hashed by one or more different hash functions. The resulting hashes are used to set bits in the bit array. To check if an item is already in the set, it is once again hashed by the same hash functions, and the corresponding bits are inspected to see if they had been previously set. This organization means that an item may be incorrectly reported as being present in the set (false positive), but may never incorrectly report that an item is not present, when it actu-

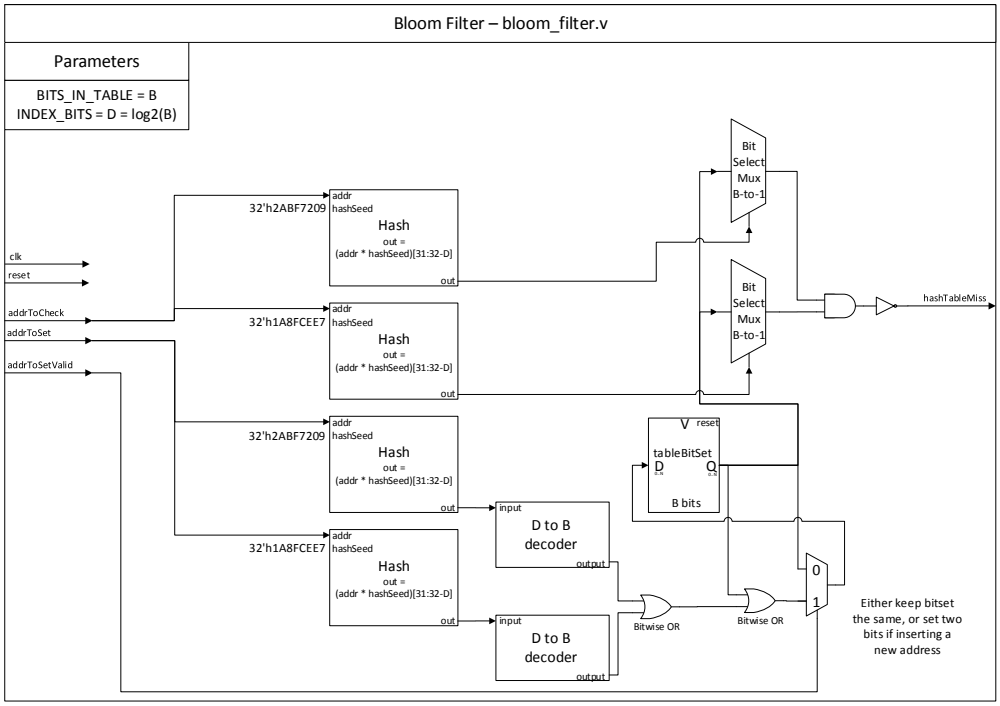



**Figure A.30:** A schematic of the bloom filter submodule. As shown here, two hash functions are used per address being checked/set.

ally is (false negative). By sizing the bloom filter bit array large enough, and by providing a sufficient number of hash functions, false positives can be minimized.

The bloom filter plays a very important performance role for the ring cache. The bloom filter module hashes every address that is being stored into ring cache, and provides an interface to check whether an address being loaded has previously been stored. If an address being loaded has previously been stored in the ring cache, but subsequently evicted, a remote load must be sent over the request/reply networks to fetch it from the L1 of the owner core of that address. If an address being loaded was never previously stored in the ring cache, it may load from the L1 of the local core that is requesting it. For some loops in some benchmarks, the number of remote loads is very high without the bloom filter. Some of those remote loads are the result of cold cache load misses. Others result from the compiler, which must be conservative in the face of memory aliases – if it can't prove an address is not shared, it must assume it is shared. This causes a number of loads to the ring cache for data that isn't actually shared, that will never actually be stored in the ring cache. The bloom filter, assuming the false positive rate is not too high, removes a significant portion of these unnecessary remote loads.

For hash functions we implemented multiply and shift hashing. Each 32-bit address is multiplied by a randomly chosen hash seed, with only the least significant 32-bits of the product kept. These resulting 32-bits are shifted right until only the required number of bits to index into the bloom filter bitset remain. The bit corresponding to the particular index is then accessed or set, depending on whether we are checking for membership or inserting an item.

We experimented to determine how large a bloom filter was required to achieve good performance. In short, we found that a one hash function bloom filter with 256 bits of storage was more than enough to reduce false positives to an acceptably low level for all of our evaluated benchmarks. Moreover, using just the cache array index bits of the address (8 in our design) was almost as good as using a proper multiply-and-shift hash function. Since the extra hardware for the multipliers in-

creases the critical path of the ring cache, we do not include them in the design. Since this result may be very specific to our particular SPEC 2000 CINT benchmarks, we still include support for the full bloom filter hardware, in the event that it is necessary for other programs. For our 6 SPEC INT benchmarks, three of them (164.gzip, 175.vpr, and 197.parser) had a significant number of remote loads without a bloom filter. Compared to using no bloom filter at all, the simple 1-hash bloom filter reduced the number of remote loads by 97%, 92%, and 60%, respectively. Without it, speedups were reduced by 68%, 48%, and 22%. False positives accounted for a negligible increase in remote loads.

The bloom filter *must* be cleared between loop invocations, or it will quickly fill up and have a very high false positive rate. Since there is a memory barrier between every loop invocation, it is safe to clear the bloom filter at that point, since every value previously stored in the ring cache will already be flushed to the normal cache hierarchy.

### Parameters

BITS_IN_TABLE The size of the bloom filter potentially has a large impact on false positive rates when checking for set membership.

INDEX_BITS Log2(BITS_IN_TABLE) is the number of bits needed to address the bitset. It also corresponds to the number of bits the hash functions should output.

### Inputs

reset This one-bit input must be set high for at least one cycle between loop invocations to clear the bloom filter of all state.

addrToCheck This 32-bit input is the address that is being checked for set membership in the bloom filter.

addrToSet This 32-bit input is the address that is being inserted into the bloom filter.

addrToSetValid This one-bit input is set high if *addrToSet* is a valid address to insert into the bloom filter.

### Outputs

hashTableMiss The single one-bit output from the bloom filter indicates of *addrToCheck* was present in the set or not.

### Datapath

A schematic of the bloom filter is shown in Figure A.30, with a 512 bit bit-array and two hash functions. Since a ring node may process a load in parallel with a store, we may want to both insert an item into the set while simultaneously checking a different address (HELIX guarantees that the address being stored and the address being loaded are never the same, so that situation does not need to be explicitly considered). This requires four hash function operations, two for the *addrToCheck* and two identical ones for *addrToSet*. For the address being checked, the bit positions output from the hash functions are read combinationally from the bitset. If one or more of the bit positions were not already set to 1, then this address has not been seen before, and *hashTableMiss* is set high. For the address being set, the calculated bit positions are used to set the two corsponding bits to 1 in the bitset at the clock edge.

### Control

There is no control for this module.

## A.12 Signal Buffer

The signal buffer contributes a significant fraction of the improved speedups that ring cache provides HELIX. It produces speedups both by pushing the signal tracking logic to hardware instead of software, and by decoupling signal forwarding from synchronization, which helps break the sequential forwarding chains of synchronization which are intrinsic to HELIX and DOACROSS style parallelization. The amount of hardware resources dedicated to the signal buffer can increase speedups along two dimensions. First, adding more available signal IDs allows the compiler to more aggressively parallelize sequential code into smaller sequential segments, potentially increasing parallelism amongst segments. Second, adding more bits for buffering received and sent signals allows cores to increase the number of iteration *epochs* they can decouple from each other during execution, which reduces core idle time that normally results from sequential forwarding chains. We start the signal buffer discussion by first explaining the concept of *epochs* in this context, and how the signal buffer facilitates synchronization decoupling by allowing cores to skip *light waits*. Next, a datapath and control FSM of the signal buffer reference design is presented. The hardware implementation of two important parameters – number of signal IDs and amount of signal buffering – is described. Then, we discuss some optimizations the compiler can make to reduce the amount of synchronization instructions that need to be sent to the signal buffer. Finally, we discuss how the signal handling used in our previous publications – a real multicore in the original HELIX paper [11], and our simulated ring cache enabled multicore in HELIX-RC [9] – map to our generalized reference design. We follow up in Section A.14 with area and timing results for the signal buffer.

### A.12.1 Synchronization Epochs

Previously, Section A.2 described at a high level how decoupling signal forwarding from synchronization allows HELIX-RC to break sequential forwarding chains and increase speedups. In this

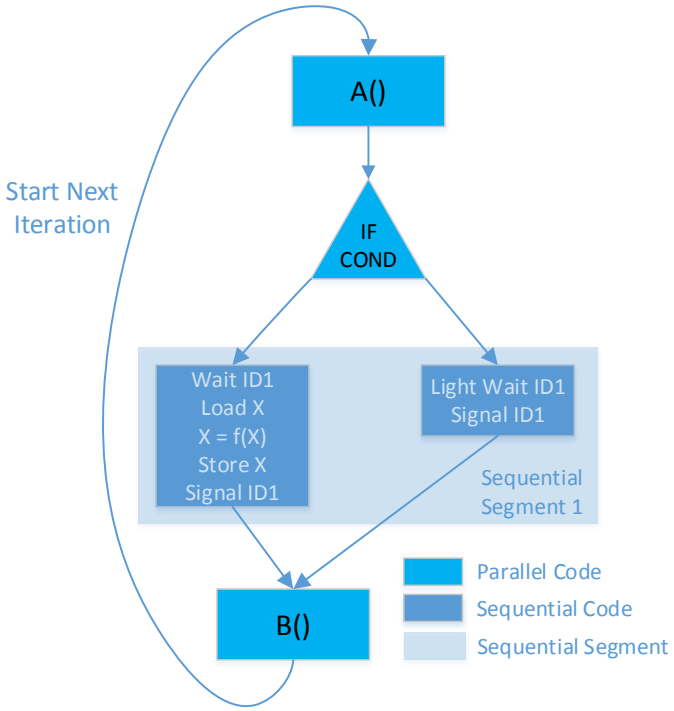



**Figure A.31:** A modified loop body where empty sequential segments start with *light wait* instructions instead of ordinary wait instructions.

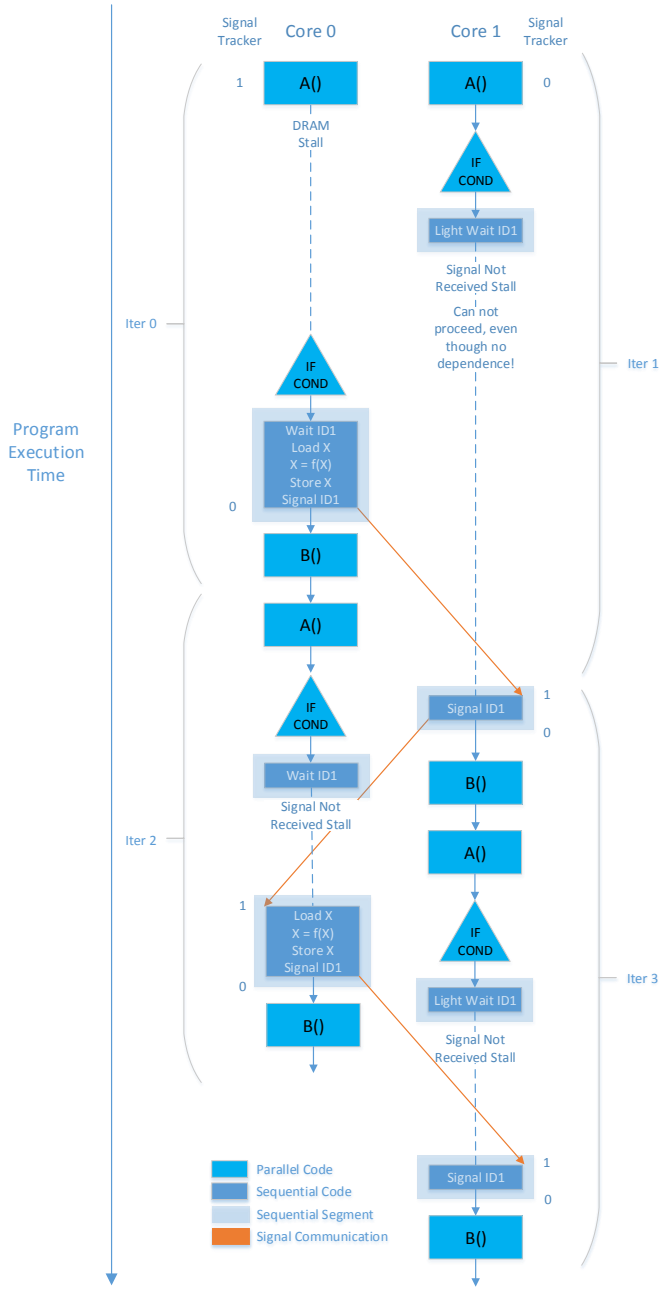



**Figure A.32:** A single bit of state for synchronizing signals constrains cores to operating within a single *synchronization epoch*. In a two core system, this implies cores can not move apart by more than 2 iterations.

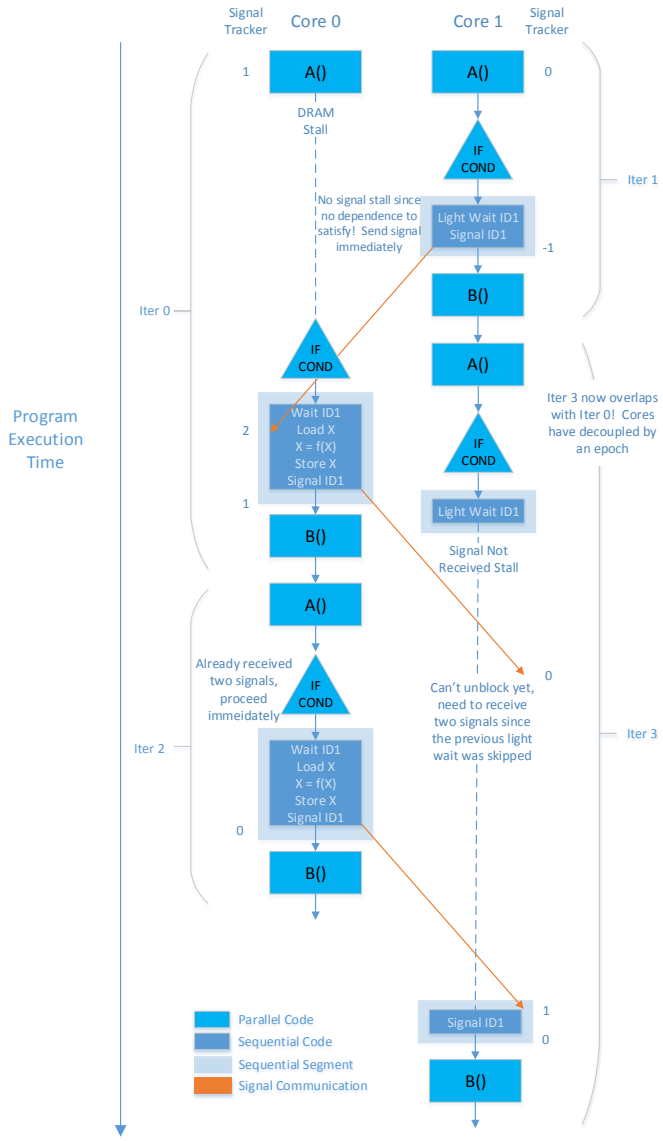



**Figure A.33:** Using two bits of signal buffering state improves performance by allowing cores to decouple an additional *synchronization epoch*.

section, we describe in detail the operation of the signal buffer, how it breaks these chains, and how it allows cores to decouple in units of *epochs*.

We define an *epoch* to be a set of N iterations, where N is the number of cores executing a parallel loop. Consider a 3-core system. For a parallel loop with 8 total iterations, core 0 executes iterations 0, 3, 6; core 1 executes iterations 1, 4, 7; core 2 executes iterations 2, 5, 8. An *epoch*, therefore is 3 iterations long. The start of an *epoch*, however, does not need to start at the beginning of a trip of iterations. For example, an *epoch* could span from iteration 1 to iteration 3, not just from 0 to 2.

We further define the concept of a *synchronization epoch*, which is an *epoch* whose bounds are not at iteration boundaries. Instead, they exist just before the next sequential segment in an iteration. Take our example of an *epoch* of iterations from 1 to 3. The corresponding *synchronization epoch* would begin precisely at the next sequential segment encountered by the core executing iteration 1, and would end just before the next sequential segment encountered by the core executing iteration 3. In higher level terms, a *synchronization epoch* is the furthest distance any pair of cores can drift apart if they are constrained by a sequential forwarding chain. In the case where all sequential segments contain dependences that need to be satisfied, no two cores can ever separate by more than a *synchronization epoch*, even if signal buffering is available, since all sequential segments must be executed in loop iteration order. However, if there are sequential segments that are empty and there is sufficient signal buffering available, cores can drift apart ("decouple") by multiple *synchronization epochs*, which reduces core idle time. We define wait instructions that mark the boundaries of such empty sequential segments as *light waits*, which do not need to be synchronized under certain circumstances.

An example is helpful to understand these two concepts. Figure A.31 presents a modified version of Figure A.2, except this time with a *light wait* instead of a normal wait instruction on the right branch of the sequential segment, indicating that the segment lacks any dependence. To keep things simple, we assume only a two core system – the same conclusions will hold true for a chip with more

cores. Each core only needs to receive signals from the other core to unblock a particular sequential segment. First, we consider a setup where only a single bit per core is used to track signals received for a particular sequential segment for each other core in the system. Since we have only two cores, and only one sequential segment, this means each core only needs a single bit total to track signals. When a core receives a signal, it sets the signal bit. When a core finishes executing the sequential segment, it clears the bit, therefore "consuming" the signal.

Figure A.32, depicts the execution of four iterations of the example loop. The time it takes to transmit a signal is exaggerated to better illustrate the impact of signal buffering. Note that since core 0 is executing the first iteration of a loop, its signal bit is preset, since there is no iteration -1 to receive a signal from. Data communication is not shown for simplicity. First, core 0 and core 1 start executing the parallel portions of iterations 0 and 1. Just before reaching the sequential segment, core 0 is stalled on a DRAM access. Meanwhile, core 1 reaches the *light wait* instruction at the start of the sequential segment. Core 1, executing iteration 1, hasn't received the signal from core 0, executing iteration 0, yet, so may not enter the sequential segment. However, since it took the right branch of the *if*, the sequential segment starts with a *light wait* rather than a normal wait. Core 1 therefore knows that it doesn't contain a dependence to synchronize, so would prefer to continue on with execution, even though it is blocked. In this situation, core 0 and core 1 are said to both be within the same *synchronization epoch*. Once the DRAM access returns, core 0 executes the segment, clears its signal received bit, and sends the signal to unblock core 1. Core 1 sets the signal received bit, which grants it access to the sequential segment, before quickly sending its own signal, which clears the received bit. Core 0 soon begins executing iteration 2, and core 1 begins executing iteration 3, where the same dynamic repeats itself, though without the DRAM stall. Even though core 1 never needs to access shared data, it is nonetheless constrained by the sequential segment. When HELIX does not have access to a ring cache, cores will always be constrained to a single *synchronization epoch*. This can be seen by observing that iteration 0 only ever overlaps with iteration 1, which in turn only

overlaps with iteration 0 and iteration 2.

Imagine a scenario where core 1, knowing that the sequential segment doesn't contain a dependence, bypasses the *light wait* instruction anyway. It would send the corresponding signal, which would ordinarily clear core 1's bit, and set core 0's bit. However, core 1's bit is already cleared, and core 0's bit is already set. In this case, the information that a signal was consumed and that a signal was received was lost. Core 1, upon receiving the signal from core 0 (executing iteration 0), will set its bit and therefore enter the sequential segment of iteration 3, even though that signal was meant for the segment it skipped in iteration 1. This potentially results in accessing shared data not in iteration order, a correctness violation. Likewise, core 0 has lost the knowledge that it received a signal from core 1 iteration 1, so may not enter the sequential segment in iteration 2. Since we would like core 1 to be able to skip the empty sequential segment, we add an additional bit to our signal tracking. Instead of 2 states (received signal or not), there are now 4. These new states allow cores to record whether they've skipped a sequential segment (state -1), and therefore need to receive two more signals to enter the next non-light sequential segment, and whether they've received an extra signal (state 2), and therefore are free to enter the sequential segment two more times.

Figure A.33, depicts a new execution timeline when this additional signal buffering hardware is added. This time, core 1 is able to skip the sequential segment and race ahead to iteration 3 without violating correctness. Core 1 takes the right branch of the *if*, and even though the sequential segment is empty once again, must block. However, because of the extra signal buffering capability, the cores have now been able to drift apart by an additional *synchronization epoch*, letting core 1 start execute iteration 3 before iteration 0 has executed the sequential segment. For every two additional states that are added to track signals, cores can drift apart yet another *synchronization epoch*. Of course, cores can only decouple if they encounter sequential segments that are empty – otherwise, since they need to access shared data, the sequential segments still need to be executed in loop iteration order. Our benchmarks contain enough empty segments, though, that decoupling cores reduces idle time

and increases speedups, significantly for some benchmarks, as we saw in Figure A.5. In our simplified example, the execution time only improves slightly. In more realistic scenarios, with more cores and more heterogeneity in execution, having cores execute as far ahead and send as many signals as soon as possible increases overall performance even more. In practice, even given unconstrained resources, the maximum that cores drift apart in our complex benchmarks is usually limited to two *synchronization epochs*, so only 4 states for signal buffering are required for most of the benefit.

### A.12.2 Signal Buffer Module

The following sections describe the datapath and control implementation of the reference design of the signal buffer. This implementation is parameterized and more general than the specific one used by HELIX-RC. After explaining the generalized design, Section A.12.6 will describe how the signal buffer from HELIX-RC [9] maps to the general design, and the specific values of the parameters. First an overview of the datapath will be presented. Then, the control for initialization and general operation will be described. Figures A.34, A.36, and A.37 present datapath schematics of the signal buffer, split into three different hierarchical levels – the signal buffer module, the signal tracker module, and the core tracker module. Each module will be described separately. Since the core tracker module contains most of the business logic, the section dedicated to that module will contain most of the control discussion. Note that the datapath schematics are meant to be an overview, and the Verilog reference design should be consulted for exact bitwidths, etc.

This outermost module contains the entire signal buffer, shown in Figure A.34. Signals and waits arrive from the core and forwarding network. Signals are recorded internally, and wait instructions are released only when release criteria are met (i.e., when enough signals have been received).

Parameters The signal buffer is impacted by three system parameters. First, the *signal bandwidth* parameter indicates how many signals the signal buffer must be able to process in a single

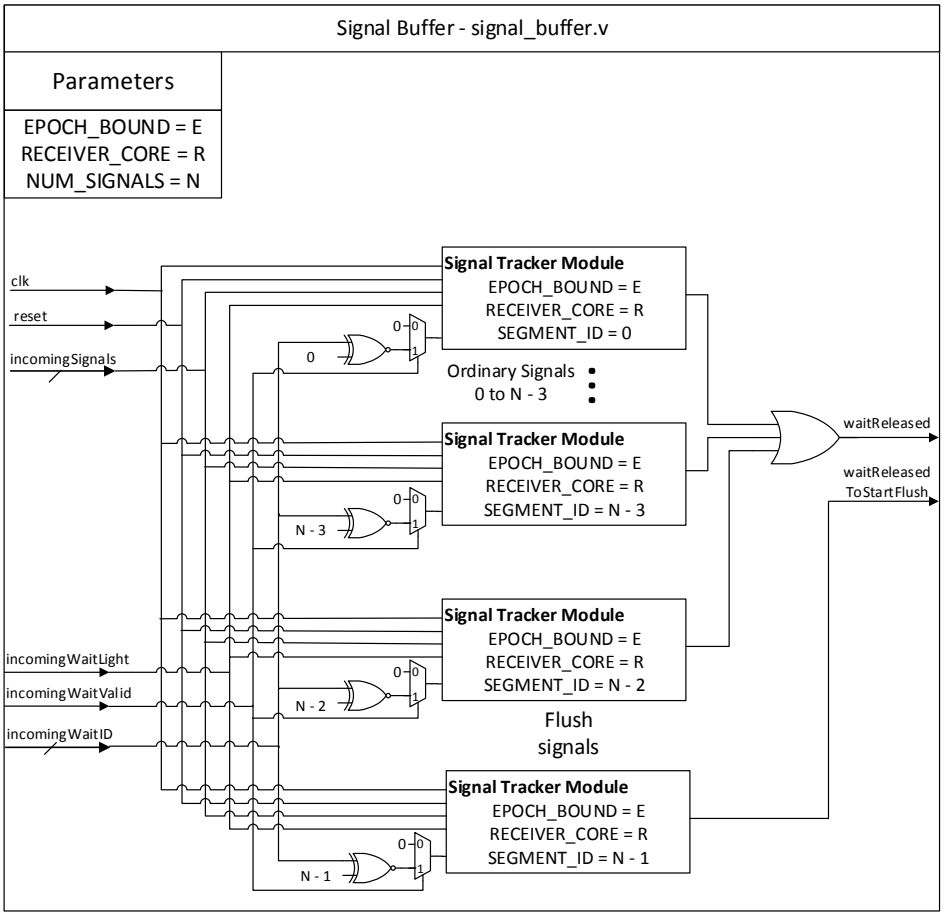



**Figure A.34:** A signal buffer module contains signal tracker submodules, one for each sequential segment that must be tracked (128 in our reference design). It processes up to *signal bandwidth* signals per cycle, as well as checking if one wait instruction per cycle can be released.


**Figure A.35:** A signal entry contains a valid bit, a core ID and a segment ID. In our reference implementation with 16 cores and 128 total sequential segment IDs, this totals 1 + 4 + 7 = 12 bits per signal.

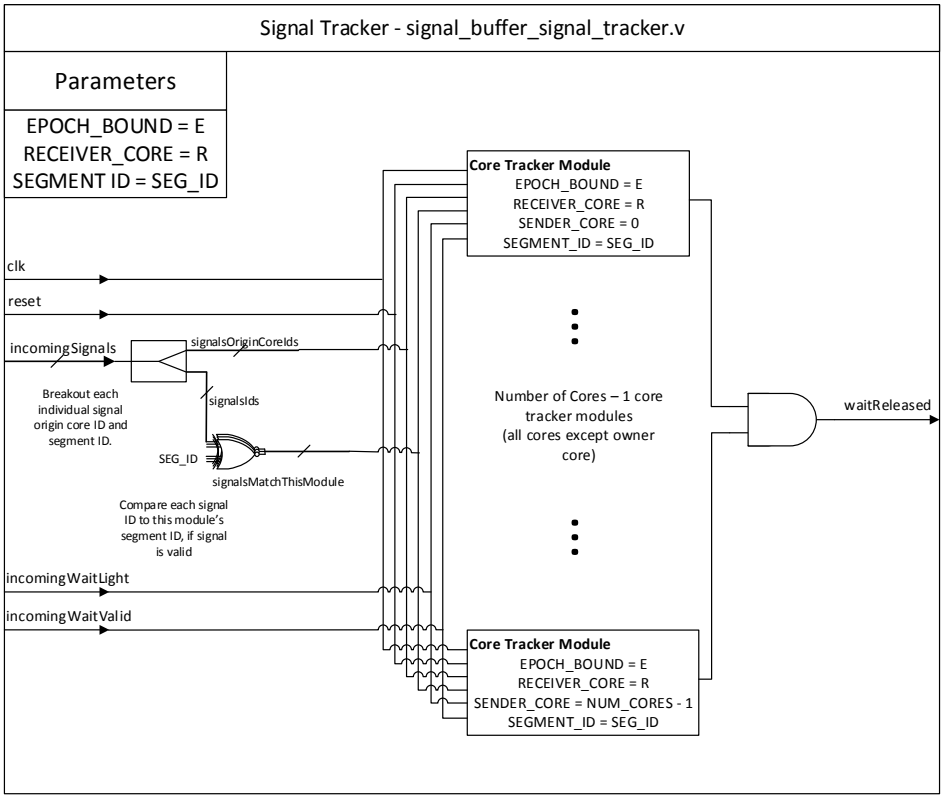



**Figure A.36:** A signal tracker module handles all waits and signals to a single sequential segment ID. It contains core tracker submodules, one for each other core besides the receiver core of this signal buffer (therefore 15 submodules on a 16 core system). It determines whether a sequential segment is safe to enter by the receiver core.

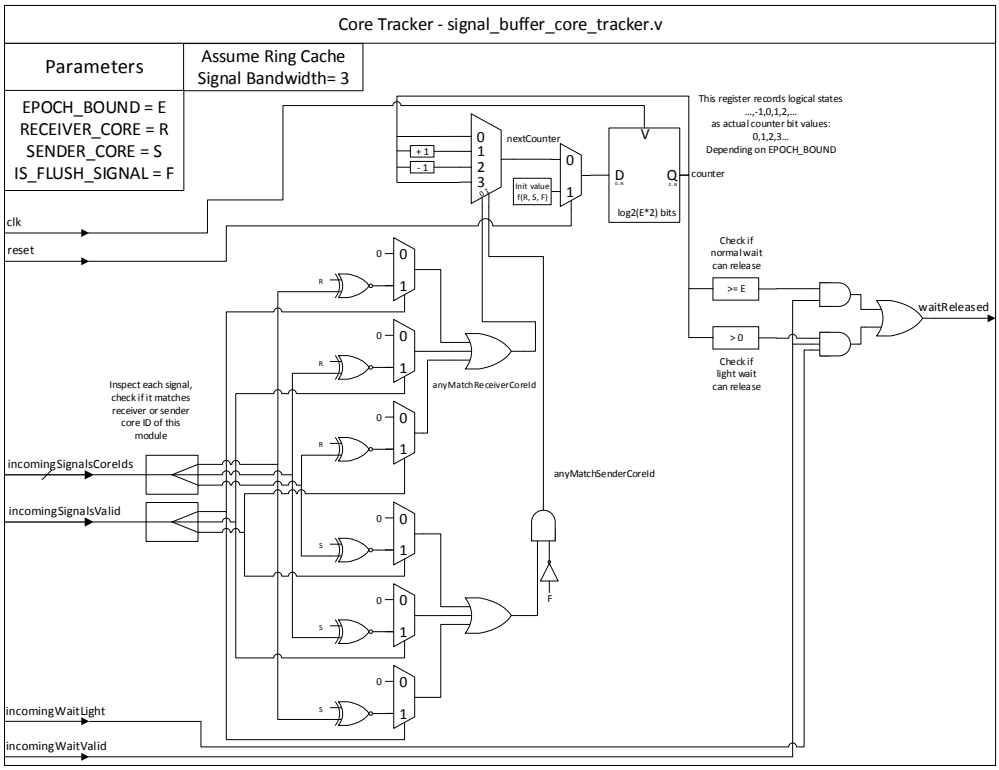



**Figure A.37:** A core tracker module handles all waits and signals to a single sequential segment ID. It contains a counter for tracking number of signals sent by a single sender core, as well as sent from the receiver core. It determines whether it is safe for the receiver core to execute a wait instruction and enter a sequential segment, as far as the single sender core is concerned.

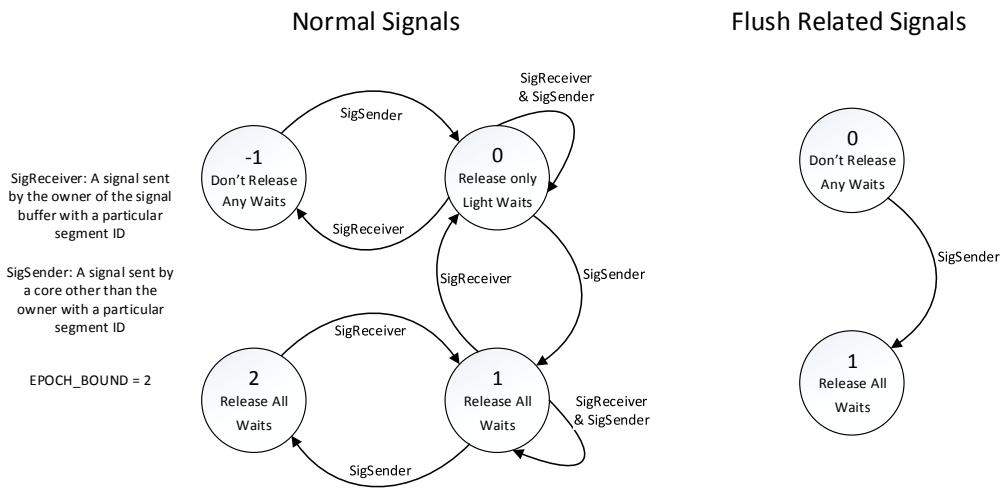



**Figure A.38:** The state machine of the core tracker module counts the difference between signals received from the sender core minus signals sent from the receiver core. Depending on the *epoch bound* parameter and the counter value, different types of wait instructions can be released. In this example, epoch bound is 2. Flush related signals require only recording the reception of a single signal.

## Normal Signals

| Signal Buffer Owner Core | Core Tracker Sender Core | Counter Initial State |
|---|---|---|
| Core 0 | Core 1 | State 1 |
| | Core 2 | State 1 |
| Core 1 | Core 0 | State 0 |
| | Core 2 | State 1 |
| Core 2 | Core 0 | State 0 |
| | Core 1 | State 0 |

## Flush Related Signals

| Signal Buffer Owner Core | Core Tracker Sender Core ID | Counter Initial State |
|---|---|---|
| Core X | Core Y | State 0 |
| | Core Z | State 0 |

**Figure A.39:** For the start of every loop invocation, every counter in the signal buffer must be initialized properly, accounting for the fact that the first epoch of iterations only need to be unblocked by cores with a lower core ID. Flush related signals are always initialized to 0.

clock cycle, which has a noticeable affect on signal buffer area. Second, the *epoch bound* parameter indicates how many *synchronization epochs* two cores can drift apart in execution, which increases the amount of bits needed to track signals. Finally, the *num signals* parameter indicates how many unique signal IDs that the signal buffer tracks. The compiler must know how many signal IDs are available prior to compilation, as it limits the total number of sequential segments it can create in any given loop. The top two signal IDs are reserved for two special signals related to the ring cache flush that happens at the end of every loop invocation.

Inputs At the highest level, the signal buffer has several different inputs pertinent to its primary operation. First, it receives one or more signals on the *incomingSignals* input, up to the maximum depending on the *signal bandwidth* parameter. Figure A.35 shows the bit layout of a signal – multiple of these are combined together on the *incomingSignals* bus. These signals may originate from other cores (*sender cores*), or from the core which contains this signal buffer (which we call the *receiver core*). Each incoming signal contains the core ID of whoever sent it, in addition to the sequential segment ID that it corresponds to. It is expected that these signals arrive sometime near the beginning of the clock period, and are all serviced by the next clock edge, at which point they are removed as inputs. The signal buffer, therefore, must be able to handle *signal bandwidth* number of signals per cycle. In addition to signal inputs, the signal buffer also has three inputs related to wait instructions. Incoming wait instructions are only issued by the *receiver core*. The first of these, *incomingWaitValid*, indicates whether a wait instruction is being executed by the core. The second, *incomingWaitID*, contains the sequential segment ID that is protected by this particular wait instruction. Finally, an additional input bit, *incomingWaitLight*, indicates whether this wait instruction is protecting an empty sequential segment, and therefore is subject to different release criteria.

Outputs There is one main output, *waitReleased*, which is set high if and only if the *receiver core* is executing a wait instruction that the signal buffer has determined is safe to release, thereby allowing the core to enter a sequential segment. A secondary output, *waitReleasedToStartFlush*, is dedicated for a special set of signal tracking bits related to the flush operation. These signal tracking bits are initialized differently than a normal signal, and releasing their accompanying wait has a special semantic to the *receiver core*, so they require a special output from the signal buffer module. The *waitReleased* outputs will be held high as long as the wait instruction release criteria are met, and the wait instruction related inputs are still valid.

Datapath Within the signal buffer, there is a submodule instantiation per signal ID that needs to be tracked. The *incomingSignals* are routed to each of the signal tracker submodules, which have been instantiated with a signal ID to track, the *epoch bound* parameter, and the ID of the *receiver core*. The *incomingWaitValid* signal is passed on to a submodule if and only if the *incomingWaitId* matches the signal ID of the particular submodule. Therefore, only one of the submodules will receive a high input on their own *incomingWaitValid* input. The *incomingWaitLight* input is also passed on to each of the signal tracker submodules. A signal tracker that receives a high *incomingWaitValid* signal will raise their *waitReleased* output to high if sufficient signals have been received from every other core, therefore allowing the *receiver core* entrance into the sequential segment. Every signal tracker submodule's *waitReleased* signals are ORed together to set the signal buffer's *waitReleased* output. Since only one one wait instruction can be executed at a time, and it is only passed on to one signal tracker submodule, then only one submodule will raise their *waitReleased* signal. There are also two special trackers dedicated to the top two possible signal IDs, which are used to facilitate the end of loop flush. These two trackers are initialized slightly differently than a normal signal tracker, and only require the *epoch bound* parameter to be set to 1.

Control The outermost signal buffer module lacks any control, instead it routes inputs/outputs to/from the signal tracker module.

### A.12.3 Signal Tracker Module

This module is responsible for tracking all signal tracking and wait releasing pertinent to a single sequential segment ID, and is shown in Figure A.36.

Parameters The signal tracker module contains three parameters set by the parent signal buffer module. The first of these is the *epoch bound* parameter as described in the signal buffer module section. Next, the module keeps track of its *receiver core* ID. The third parameter, *segment id* is the sequential segment ID that this module is responsible for.

Inputs The inputs are the same as those of the signal buffer module, with the exception that *incomingWaitValid* is only set high by the parent module if the wait ID matches the sequential segment ID that this module instantiation is responsible for (*segment id*). The timing of all of the input signals is identical to the parent module.

Outputs The single output is *waitReleased*, which is set high if and only if the *receiver core* is executing a wait instruction whose ID matches *segment id*, and enough signals have been received that the core may enter the sequential segment. This latter criteria is met only when the *waitReleased* output of every core tracker module is set high. The output is held high as long as the wait related inputs are valid and the release criteria are met.

Datapath The signal tracker module contains N-1 instantiations of the core tracker module, where N is the total number of cores in the chip. There are only N-1 of these modules since the *receiver core* doesn't track itself. Each of the core tracker submodules are assigned a core ID corresponding to each possible sender core. A special parameter is set if the submodule corresponds to a

special flush signal. At this level, the signal tracker module is only concerned about incoming signals that match the ID of the signal tracker itself. Some minor logic is used to set an array of valid bits for the incoming signals, where the valid bit corresponding to an incoming signal is only set if its signal ID matches that of the signal tracker. The IDs of the cores that sent the signals are also broken out from the *incomingSignals* bus. The reason that all signals with a matching ID are sent to the submodules, rather than just those that match the signal ID and the sender core ID, is that signals sent from the actual *receiver core* must also update every core tracker submodule. The signal valid bits and sender core IDs are sent to all of the core tracker submodules, in addition to the *incomingWaitValid* and *incomingWaitLight* inputs.

Control This module lacks any control, and primary just routes inputs/outputs to/from the core tracker modules.

### A.12.4 Core Tracker Module

This module, shown in Figure A.37, is responsible for tracking signals received by an sent to a single sender core, for a single sequential segment ID.

Parameters As with the parent and grandparent module, the core tracker module contains an *epoch bound* parameter. It also contains a *receiver core* ID parameter, and a *sender core* ID parameter. There is a fourth parameter, *special flush signal* to indicate that this module is for tracking signals to one of the reserved signal IDs pertaining to the end of loop flush. For such a special signal, the *epoch bound* parameter is always set to 1, since the special signals are only sent once at the end of a loop invocation, and therefore don't have epochs to track.

Inputs The *incomingSignals* bus from the parent module has been split into an *incomingSignalsValid* array of bits and an *incomingSignalsCoreIds* bus. The former indicates which of the sig-

nals received by the signal buffer matched the signal tracker module segment ID which contains this core tracker module. The latter indicates which core IDs sent which of the incoming signals, one per *signal bandwidth* supported by the system. There are also *incomingWaitValid* and *incomingWaitLight* inputs that are identical to those of the parent module. Input timing is identical to the parent module.

Outputs The output of a core tracker, *waitReleased*, is set high if the *receiver core* has received enough signals from the *sender core* to enter the sequential segment. It is held high for as long as the wait related inputs are valid, and the release criteria are met.

Datapath The core tracker module is where the bulk of the signal buffer work is performed. Within each core tracker module, there is a counter, where the number of states needed to be represented by the counter correspond to two times the number of *synchronization epochs* that we allow cores to decouple by. There can be at most two valid incoming signals for each core tracker – one sent by the *sender core*, and one sent by the actual *receiver core*. Since signals are sent and propagated around ring cache in-order, and without passing each other, it is impossible to receive two copies of a signal from a single core with the same signal ID. Depending on which of these two possible signals are received by the core tracker, the internal counter is either incremented, decremented, or held to the same value. Depending on the value of the counter, and whether *incomingWaitValid* and/or *incomingWaitLight* is high, the only output, *waitReleased*, is raised high. This indicates that as far as this sender core is concerned, the *receiver core* is free to execute the sequential segment associated with this particular wait instruction.

Control Now that that datapath and module hierarchy is established, the control and logic governing how the core tracker counters are updated is described, in addition to the conditions in which waits and light waits are released. As previously described, a particular core tracker module can re-

ceive at most two signals for a particular signal ID on a given cycle – one from the *receiver core* of the signal buffer, and one from the sender core corresponding to the particular instantiation of the core tracker module. We can call these two signals SigReceiver and SigSender. Processing a SigSender has the semantic that the sender core has exited the coresponding sequential segment. Processing a SigReceiver has the semantic that the *receiver core* has just exited the sequential segment. The counter in each core tracker module keeps track of the difference between the number of SigReceivers and SigSenders processed. Figure A.38 depicts the FSM for how these two signal types get processed. The *epoch bound* is set to be 2 in this implementation, so the number of counter states required is 4. Whenever a core tracker processes a SigSender, the counter is incremented, indicating that the sender core has executed a particular sequential segment. Whenever a core tracker processes a SigReceiver, the counter is decremented, indicating that this *receiver core* has consumed one of the received signals by finishing execution of the sequential segment. If *incomingWaitValid* is raised high but *incomingWaitLight* is low, the core tracker sets *waitReleased* high if and only if the value of the counter is greater than or equal to 1. If both *incomingWaitValid* and *incomingWaitLight* are raised high, then waitReleased is set high only if the counter value is greater than the minimum possible (-1 in this configuration). Note that although we refer to states ranging from -1...2, in hardware the counter register will range from 0...3. We refer to the logical states to better map to the idea of cores being in different relative epochs.

The natural question is, what do the different counter values represent? What is the semantic of the different states? How come light waits are released at a different threshold than normal waits? Consider the case where only normal wait instructions are being executed (i.e., every sequential segment contains dependences that must be satisfied in loop iteration order). In this situation, as discussed in section A.12.1, cores can not decouple by more than one *synchronization epoch* in order to properly satisfy the dependences. In that situation, state 0 refers to a *receiver core* that has not yet received a signal for a particular segment this *synchronization epoch* – this core is further along in exe-

cution than the corresponding *sender core*. The *receiver core* may be executing parallel code, or may be stuck waiting on a wait instruction, waiting to receive a signal. In either case, since the upcoming segment contains a dependence, it can not execute it until it receives a signal from every other core. State 1 in this situation refers to a *receiver core* that has received a signal for a particular segment, but has not yet executed (and therefore consumed the signal) of that sequential segment yet – this core is further behind in execution than the *sender core*. When this core executes a wait instruction, it knows it has received a signal from the corresponding sender core, which implies that the sender core has exited the sequential segment. The *receiver core* executes the segment and then sends a SigReceiver, which reduces the counter to state 0. As long as only non-light wait instructions are executed, the counter can only be in state 0 or state 1, as was the case in Figure A.32.

However, if light wait instructions are executed, it implies that a core is executing a segment that does not actually need to be synchronized. Imagine a *receiver core* whose counter is currently in state 0. Now it encounters a light wait instruction, which it can skip and send the SigReceiver signal immediately. This reduces its state to -1, indicating that not only was it already ahead of the corresponding *sender core*, but now it is ahead by an additional *synchronization epoch*. It now needs to receive two signals from the sender core in order to unblock the next non-light wait instruction it encounters. The sender core, upon receiving the signal from the *receiver core*, increments its state from 1 to 2, indicating that it has two outstanding signals to consume, as was the case in Figure A.33.

Notice that in the FSM that there aren't any transitions representing either underflow or overflow of the counter. In our previous example, what would happen if the *receiver core* were to race ahead an execute the sequential segment from yet another *synchronization epoch* ahead? Absent any back pressure, it would skip the light wait instruction, send the signal, underflowing its counter, which would be received by the *sender core*, overflowing its counter. This is the reason that light wait instructions must still be able to block a core, and is the reason why a light wait may only be released by the signal buffer if and only if the counter is not about to underflow. If a *receiver core* is

prohibited from underflowing its counter, then no *sender core* can ever overflow. This is a result of the fact that counter states are symmetric – a state of -1 indicates a core is ahead by one *synchronization epoch*, and a state of 2 indicates it is behind by one. If a core is prevented from moving forward, past the hardware limit, to an *epoch* too far in the future, then this implies that no core can ever fall behind an additional *epoch* past the hardware limit. If the *epoch bound* parameter is increased, then cores can decouple even further without blocking due to hardware limits.

Figure A.38 depicted the control for a single core tracker module. Remember that to release a wait instruction from a *receiver core*, all (numCores - 1) core tracker modules must agree to release the wait. For a normal wait, this indicates that all sender cores have executed the sequential segment from iterations older than the receiver. For a light wait, this indicates that no signal buffer counters will overflow or underflow if the light wait is released.

If a core tracker module is designated special by the *special flush signal* parameter, the control is slightly modified. Since these core tracker modules are merely tracking whether every core has executed a particular special signal just once, any SigReceivers are ignored, since *epochs* are not relevant (*epoch bound* is set to 1).

Initialization The state counters are initialized at the beginning of every loop in a particular way, depending on the *receiver core* ID and the *sender core* ID. For the first *synchronization epoch* of a loop, each core expects a different number of signals from all of the other cores. Consider a 3 core system where core 0 always runs iteration 0 of a loop. Figure A.39 depicts the state of each core's signal buffer just as a new loop begins. Since iteration 0 is the first in a loop, it doesn't need to be unblocked by any core for core 0 to enter any sequential segment. Therefore, all of the counters for every signal tracker in core 0's signal buffer are initialized to state 1 at the beginning of every loop, indicating that all segments can be entered without receiving any signals. Core 1, on the other hand, when executing iteration 1, needs to receive signals from iteration 0 to enter any segment. Conse-

quently, the core tracker modules corresponding to core 0, for every signal tracker in core 1's signal buffer, are initialized to state 0, indicating that signals must be received by core 0 before entering any segments. The counters corresponding to core 2 in core 1's signal buffer, on the other hand, must still be initialized to state 1, since core 2 never executes any older iterations than core 1. Finally, every counter in core 2's signal buffer is initialized to state 0, since core 2 must receive signals from both core 0 and core 1 before entering any segments. In the case of any core tracker modules contained within a signal tracker module corresponding to special flush signal, all counters are initialized to state 0, since we desire that every core waits for every other core before releasing the corresponding wait instruction.

### A.12.5 Signal Buffer Optimizations

There are a few possible signal buffer optimizations that may be appropriate to use with the reference design. The first of these is only applicable when *epoch bound* is equal to 1 – that is, cores are not able to decouple, and must always stay within the same *synchronization epoch*, even in the presence of sequential segments that lack dependences. This is a similar scenario to when HELIX is running on a traditional multicore, albeit still with faster signal propagation speed. In this configuration, every sequential segment is run in loop iteration order, without exception. Light waits don't exist, since there is no situation where they can be skipped. The requirement for (numCores - 1) core tracker modules per signal tracker module is no longer necessary, since receiving a signal for a particular sequential segment ID from a particular core implies that every previous loop iteration from every other core has already executed the segment. This allows the signal buffer to reduce the number of core tracker modules from (numCores - 1) per signal tracker module to just 1 per signal tracker module. This reduces the total amount of signal buffer area by potentially a factor of the number of cores. The area savings comes at a cost – cores can not decouple across *synchronization epochs*, so speedups are reduced, as we saw previously in Figure A.5. However, if the signal buffer area is pro-

hibitively large, limiting *epoch bound* to 1 is a possible solution. Also, signals do not need to circulate around the entire ring any more, they only need to travel to their subsequent core in the ring, which reduces signal bandwidth requirements.

The second optimization is applicable only when *epoch bound* is equal to 2, which implies cores can decouple an additional *synchronization epoch*. If the compiler can guarantee that there is at least one non-light wait instruction per loop iteration (as would be the case if a particular dependence always needed to be satisfied), then light waits can be removed entirely, leaving only the corresponding signal behind. This non-light wait instruction *must* belong to the same sequential segment every iteration. A core can just straight away send the corresponding signal without relying on the light wait to prevent underflow. This is because the existence of at least one unavoidable non-light wait per iteration per iteration naturally limits synchronization decoupling to less than two *synchronization epochs*. This optimization is a trade-off – light wait instructions can be removed, which reduces the number of instructions executed, and the signal buffer no longer needs to contain the logic to examine light waits. However, the compiler may now have to artificially make a wait instruction non-light which otherwise could be light. In practice, we found that this downside doesn't happen often.

The third optimization applies when *epoch bound* is equal to 3. Like with the previous optimization, if the compiler can make a guarantee about non-light waits, than light waits don't need to be executed at all. Except in this situation, the compiler only needs to guarantee that at least one sequential segment executes a non-light wait per iteration. Unlike the previous case, it doesn't need to be the same sequential segment every iteration. This prevents our 6-state core tracker counters from ever underflowing. For our particular benchmarks, increasing the *epoch bound* past 2 did not have any effect on performance, so this optimization may not be useful.

### A.12.6 Previous Implementations

#### Original HELIX-RC Signal Buffer

The previously presented design is a generalized reference design, and not the exact design we used in the HELIX-RC paper. In that work, we used a specialized implementation which was designed before the generalized design was realized. Instead of a state counter and the notion of epoch bounds, we used two distinct bits per signal per *sender core*, called past and future. The second optimization from Section A.12.5 was used, so light wait instructions didn't need to be executed at all. Like the generalized design, bits were set when a *receiver core* received signals from any sender core. If the past bit was not yet set, it was subsequently set. If the past bit was already set, the future bit was now set. Unlike the generalized design, signals sent by the *receiver core* were not recorded locally in any way. Instead, a wait instruction was released when at least one bit was set, which then cleared the bit (future bit first). If a signal was received when both bits were set, the future bit was cleared – in doing so, the *receiver core* was inferring that an epoch had elapsed. This scheme had the downside that in addition to being harder to reason about, wait instructions altered ring cache state, so could not be issued speculatively. It otherwise had the same performance, effective FSM, and decoupling ability that the generalized signal buffer design has. There is no reason to use the original design over the generalized design. For the HELIX-RC work, *signal bandwidth* was set to be 5 signals per cycle (less reduced performance for `164.gzip`). The number of signals needed to be tracked by the buffer was larger than 100, since the compiler could not limit the number of sequential segments at that point in time.

#### Normal Multicore Signal Handling

On a real multicore, HELIX handled signal tracking by allocating a special private region of memory per core. Signals were implemented with stores, and waits with loads. In effect, this would map to

a signal buffer with an *epoch bound* of 1 with the first optimization in Section A.12.5. It would be unreasonable to emulate a signal buffer with higher *epoch bound*, since instead of each core only needing to write signals to a single other core's private signal tracking memory, they would need to both read and write from every other core's. The read would be necessary to either increment or decrement state, depending on the current value stored there. This explosion in cache coherence traffic would quickly dwarf any performance improvement.

## A.13 OS/Multiprogramming Considerations

Our reference design and all of our previous studies have assumed that only a single HELIX process can run at any single point in time. Currently, there is no direct support for context switching. The only time during execution when it is safe to context switch is between parallel loop invocations, after the ring node memory has been flushed. At that point in time, all states within the ring cache are default states. The only thing that needs to be done is to raise the `reset` signal, which reinitializes the signal buffer and invalidates the cache array. That procedure is the same regardless of whether the next loop to be executed is from the same process or not.

If a ring cache needs to be able to handle context switches, there are a few things to consider. First, the contents of the ring node memory need to either be tagged with address space IDs or flushed to the normal cache hierarchy during every context switch. The same mechanism for the normal ring cache flush, as described in Section A.8, can be used to accomplish this at any point in time, as long as a core does not execute any instructions after the special flush signals. All of the signal tracking bits in the signal buffer must be saved as well. This amounts to 32 bits per signal ID per core, assuming an epoch bound of 2 and 16 cores in the system.

In the current implementation, a safe way to bring the ring cache to a halt in preparation for a context switch is as follows. First, all instructions that a core has presented to the ring node interface, besides `wait`s, must be held there until the ring node indicates they have completed. This may

be hundreds of cycles in the worst case, as a number of remote loads may be pending. Waits should be removed from the ring node interface immediately, since they have no side effects on any state and will never return if their corresponding signal is not already in flight. After this last store/signal/load has finished (or a `wait` instruction has been removed), the core should not execute any more instructions from the loop. Instead, the flush mechanism of Section A.8 should be initiated by executing the special `wait` and `signal` instructions. This will guarantee that all outstanding (non-flush) store/signal activity has stopped before initiating the flush of the ring node memory in each core. It is also now safe to back up the contents of the signal buffer, through some yet to be designed mechanism. Once the flush of both the memory and the signal buffer has completed, the core's `reset` signal should go high to finish clearing the memory contents and the bloom filter. To resume execution, the signal buffer contents of each ring node need to be restored and a synchronization barrier executed. Now each core can resume where it left off.

## A.14 Synthesis Results

### A.14.1 Reference Design

This section presents some preliminary area, power, and timing results for the ring cache. Table A.1 shows the parameters for our reference design. The values were chosen to roughly match the simulated ring cache from the HELIX-RC paper. The most noticeable exception is the ring cache memory, which was 8-way set-associative rather than direct mapped in the HELIX-RC paper. Later simulations showed that the difference between 8-way and direct-mapped memory was minor, so for ease and clarity of implementation, the latter was used. It is important to understand that many of these parameters were selected to be just big enough so as not to be a bottleneck for the 6 SPECint 2000 benchmarks we evaluated. Other programs may have vastly different requirements, so these particular values should not be overly relied upon for a final implementation.

**Table A.1:** Ring Cache parameters for the reference design.

| | |
|---|---|
| Number of Supported Cores | 16 |
| Address Width | 32 bits |
| Data Width | 32 bits |
| Cache Associativity | direct mapped |
| Cache Data Storage | 1024 KB |
| Total Number of Signal IDs | 128 |
| Signal Bandwidth | 5 signals per cycle |
| Signal Buffer Epoch bound | 2 |
| Store Bandwidth | 1 per cycle |
| Forwarding Network Total Wires | 129 bits |
| Request Network Total Wires | 37 bits |
| Reply Network Total Wires | 37 bits |
| Bloom Filter Configuration | 256 bit-array, 1 hash function |
| Assumed Network Link Latency | 0.5 ns |

The reference configuration was exhaustively tested with test vectors from our X86 cycle-level C++ simulator, XIOSIM [33], with the modeled ring cache configured the same as the reference design. Vectors were collected for every Simpoint phase of all 10 of the SPEC benchmarks we evaluated in our previous HELIX-RC paper. Specifically, for every simulated cycle, the value of all inputs and outputs corresponding to the ring cache interface (Section A.6) were collected. Verilog simulations were performed with 16 ring nodes that were excited with these test vectors. At every cycle, the outputs of the ring nodes were compared to the known correct outputs from XIOSIM. Every phase passed the testbench.

Many of the parameters, such as the number of supported cores, signal bandwidth, signal buffer configuration, and total number of signal IDs, have a large impact on the area/performance of ring cache. After generating initial results for this reference design, some of these important parameters were swept. First, the reference design was synthesized using Synopsys Design Compiler and a 40nm process technology. The synthesis tool was steered to optimize for critical path delay. We used RTL-level activity factors from our SPECint test runs to more accurately predict power in the synthesized

design. Table A.2 summarizes the post-synthesis results for a single ring node. Although we assumed a 0.5 ns link latency for all of our inter-node links, we stress that the power and area numbers here do not account for any link area/power and strictly represent only Design Compiler's post-synthesis estimates. Using RTL simulation activity factors instead of Design Compiler's default, the power is reduced from nearly 100 mW to 19.22 mW because although the SPECint benchmarks use the ring cache relatively regularly, it is still only accessed in sequential segments. Since there are still significant portions of parallel code, the ring cache is often dormant.

**Table A.2:** Synthesis results for a single reference ring node.

| | |
|---|---|
| Area | 0.272 sq mm |
| Dynamic Power | 19.22 mW |
| Leakage Power | 3.3 mW |
| Max Frequency | 1.11 GHz |

Critical Path The post-synthesis timing report exposes two primary critical paths in the ring cache design. The first path starts in the forwarding network receive buffers. It continues through the bundleizer module, where stores/signals from the core may be added to the network bundle. Then it travels through the stopper module, where stores/signals may be removed from the bundle. Finally, it exits the stopper module and begins propagating over the network link to the subsequent ring node. Since link propagation accounts for 0.5 ns of the path, the other bundleizer and stopper logic accounts for only a small portion of the total path. It is beneficial that link propagation can happen in parallel with writing to the memory, since the next longest paths involve the memory. Specifically, they start at the forwarding network receive buffer and travel through the bundleizer as before. Then the path changes and instead goes to the memory array to perform a tag lookup and prepare a memory write for the next clock edge.

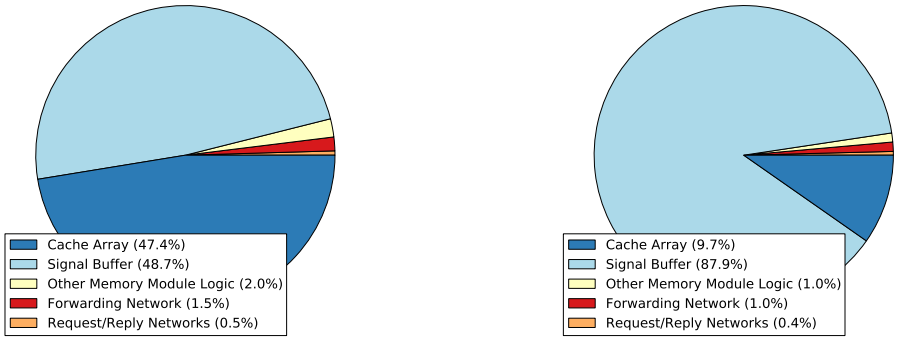



**Figure A.40:** Power and area for a single ring node. The forwarding network includes the corresponding network buffer, bundleizer, and stopper modules. The request/reply networks include the network buffers and the load unit.

Area Figure A.40a depicts the area usage in the reference design. The cache array is marginally smaller than the signal buffer. Although perhaps unexpected, this follows from the fact that the reference design uses 128 total signal IDs. Since the storage per ID is 2 storage bits per core per signal, the total number of registers per signal buffer is 2 * 16 * 128 = 4096 bits. The area of the memory is somewhat larger than it could otherwise be, since the reference design uses a register array in lieu of an SRAM to minimize access latency. If the area is prohibitively large, experimenting with using an SRAM (which would necessitate adjusting the FSMs in the memory and array modules) and reducing the number of signal IDs could provide a large benefit.

Power Figure A.40b shows the dynamic power breakdown. Although the cache array constitutes a large fraction of the area, it constitutes a smaller proportion of the power consumption. The signal bandwidth (5 signals per cycle) was set at a high level relative to store bandwidth (1 store per cycle) because the large amount of empty sequential segments tends to produce far more signals than shared data. As a result, the signal buffer is utilized far more often than the cache array.

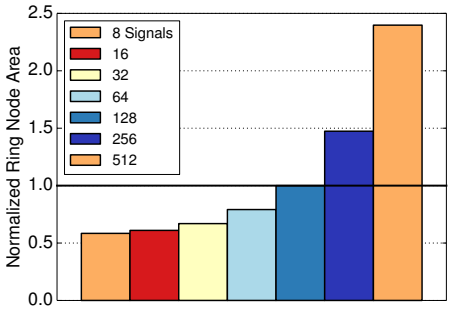



**Figure A.41:** Total ring node area as total signal ID capacity is swept from 8 to 512.

### A.14.2 Signal Buffer Parameter Sweeps

The signal buffer has several parameters that potentially have a large impact on system performance and ring node area. These include the total number of possible signal IDs, the amount of signal bandwidth, the amount of allowed synchronization epoch decoupling, and the number of cores in the system. These properties were discussed in Section A.12.

#### Number Of Signal IDs

The compiler has control over the maximum number of sequential segments it will produce in any given loop. Maximum performance can be achieved when the compiler has total flexibility to create as many sequential segments as it would like. If restricted, the compiler must combine multiple sequential segments into one, which may have an impact on performance. However, the number of signal IDs has a linear effect on the size of the signal buffer area, as each ring node must contain signal tracker modules for each possible unique signal ID. In our HELIX-RC paper, the number of signal IDs was unrestricted, which required a maximum of approximately 128 signal IDs for most loops. Due to limitations in the compiler, we are currently unable to sweep maximum signal IDs and we therefore leave this analysis for future work. However, although we can't examine it thoroughly, the intuition we have gathered from examining the compiled code suggests that significantly

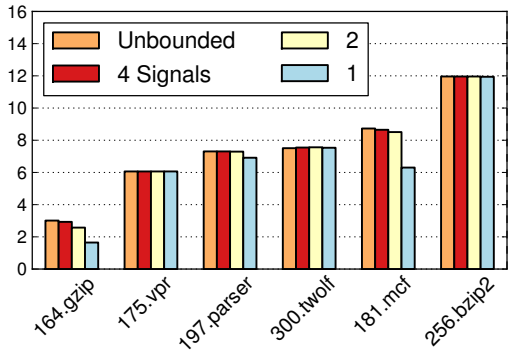



**Figure A.42:** Some benchmarks are very sensitive to signal bandwidth, with 5 signals per cycle required to maximize speedups.

fewer than 128 signals would be required to capture most or all of the performance. Some of the sequential segments are purely for edge cases (exceptions, error handling) that do not occur in normal operation and are rarely, if ever, synchronized.

We can, however, sweep the maximum amount of signal IDs in the signal buffer to see how the area changes. Figure A.41 depicts the area of a ring node, normalized to our reference design, as the number of signals are swept from 8 to 512. Given that the signal buffer was a significant fraction of the total ring node area in the reference design, it is no surprise that the total ring node area increases dramatically for the largest signal capacities.

### Signal Bandwidth

Figure A.42 from our HELIX-RC paper shows the importance of high signal bandwidth for achieving good speedups on SPECint. Although it doesn't have as drastic an effect on area as the number of signal IDs, increasing the signal bandwidth does increase the area of the signal buffer. Additional multiplexers and logic to process multiple incoming signals result in a 28% decrease in total ring node area from our reference design to one with a bandwidth of 1 signal per cycle, as seen in Figure A.43. Since the signal buffer is approximately 50% of the design area, this corresponds to about

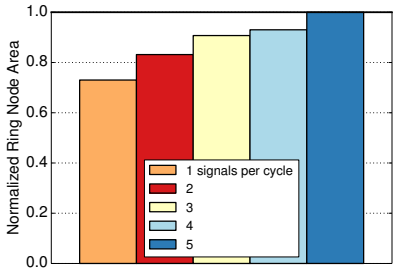



**Figure A.43:** Increasing signal bandwidth increases the signal buffer and network buffer sizes.

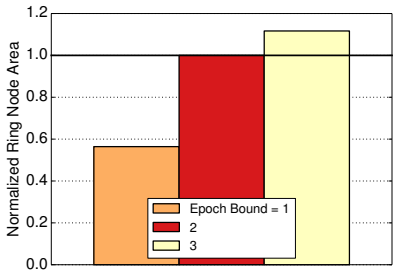



**Figure A.44:** Decoupling synchronization for one or two epochs increases area significantly, but also increases performance.

a 55% decrease in signal buffer area. Although the forwarding network buffers are also halved, their overall impact is very slight, given their size. Since the critical path involves the forwarding network buffers and the bundleizer module, the maximum achievable frequency increases slightly as signal bandwidth decreases, by around 5% from 5 signals per cycle to 1 signal per cycle.

### Amount of Synchronization Decoupling

The *epoch bound* parameter in the signal buffer dictates how many synchronization epochs cores can decouple. In the HELIX-RC paper, this parameter was essentially set to 2, with a slight optimization (see Section A.12.6). Increasing this parameter has the potential to increase speedup, but will also increase the area consumed by the signal buffer, as more bits are needed to track the state of each signal. Figure A.44 shows the area impact for three values of epoch bound: 1, 2, and 3. Note the large

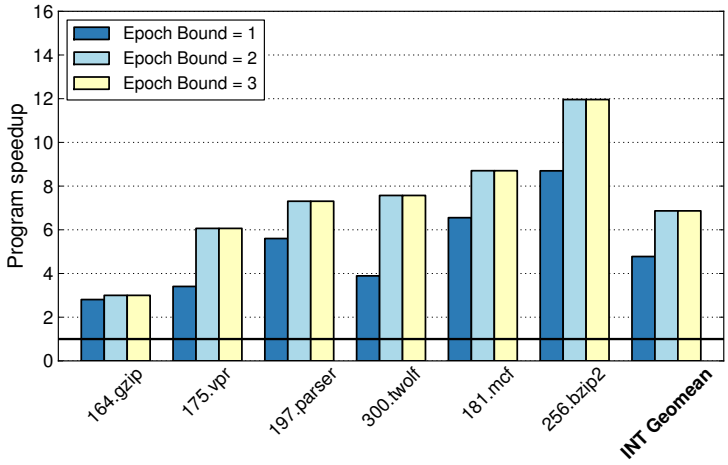



**Figure A.45:** Decoupling synchronization up to two epochs increases speedups, but beyond that there is no effect.

impact of moving from 1 to 2. This is because at a value of 1, the signal buffer can be simplified by having each core track only received signals from its immediate predecessor and send signals only to its immediate successor. This optimization is possible because at a value of 1, cores are unable to decouple. This implies executing all sequential segments strictly in loop iteration order, which removes the point of having a core broadcast a signal to every other core—since receiving a signal from your immediate predecessor guarantees that all previous iterations have already executed older iterations. However, an epoch bound of 2 has a drastic performance impact, as seen in our simulation results in Figure A.45 (this consists of some of the same data as Figure A.5). In contrast, moving to an epoch bound of 3 has absolutely no benefit—there are only a very few times in all of the combined SimPoint phases where any benchmarks decouple by that amount. Unless other program characteristics are significantly different from those of SPECint, it seems pointless to use any epoch bound value other than 2.

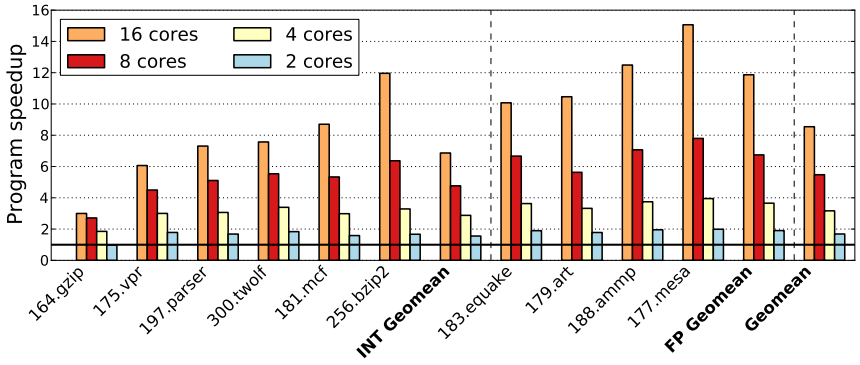



**Figure A.46:** HELIX-RC scales relatively well on a small number of cores.

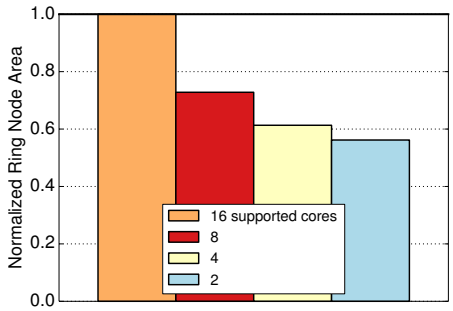



**Figure A.47:** Signal buffer size is linear with the number of supported cores, so decreasing the number of cores has a large impact on ring node area.

## Number of Supported Cores

The signal buffer needs to track received signals from every other core in the design. Consequently, the size of the state in the signal buffer varies linearly with the number of cores in the system, much in the same way that the total number of signal IDs does. Figure A.47 shows the drastic decrease in ring node size when the number of cores is reduced from 16 to 2. The size of the signal buffer decreases by a factor of 8×. Intuitively, as the number of cores decreases, so does the achievable speedup, as we previously discussed in the HELIX-RC paper and as shown in Figure A.46.

# B

# Ring Cache Verilog Code

## B.1 DEFINES.V

```
`ifndef DEFINE_V
`define DEFINE_V

`define ADDR_WIDTH 32
`define DATA_WIDTH 32

`define ID_WIDTH 7
`define NUM_SIGNALS 128 // Must be exaxctly 2^id_width

`define SIGNAL_BANDWIDTH 5

`define CORE_ID_WIDTH 4
`define NUM_CORES 16 // Must be exactly 2^core_id_width

`define TYPE_WIDTH 2
`define TYPE_LOAD 2'd0
`define TYPE_STORE 2'd1
`define TYPE_WAIT 2'd2
`define TYPE_SIGNAL 2'd3

`define SIGNAL_ENTRY_WIDTH (1 + `CORE_ID_WIDTH + `ID_WIDTH) // 1 is valid bit, CORE_ID is the origin core
`define STORE_ENTRY_WIDTH  (1 + `CORE_ID_WIDTH + `ADDR_WIDTH + `DATA_WIDTH) // ditto
`define REMOTE_LOAD_REQUEST_ENTRY_WIDTH  (1 + `CORE_ID_WIDTH + `ADDR_WIDTH)
`define REMOTE_LOAD_REPLY_ENTRY_WIDTH (1 + `CORE_ID_WIDTH + `DATA_WIDTH)

`define FORWARD_NETWORK_BUNDLE_WIDTH  ((`SIGNAL_ENTRY_WIDTH * `SIGNAL_BANDWIDTH) + `STORE_ENTRY_WIDTH)
`define REQUEST_NETWORK_BUNDLE_WIDTH (`REMOTE_LOAD_REQUEST_ENTRY_WIDTH)
`define REPLY_NETWORK_BUNDLE_WIDTH    (`REMOTE_LOAD_REPLY_ENTRY_WIDTH)


// Assume L1 line size is 64 bytes, used for address ownership calculation.
// Words on the same cache line must have the same owner
`define L1_LINE_SIZE_LOG_2 6

// Functions
`define CLOG2(x) \
(x <= 2) ? 1 : \
      (x <= 4) ? 2 : \
      (x <= 8) ? 3 : \
      (x <= 16) ? 4 : \
      (x <= 32) ? 5 : \
      (x <= 64) ? 6 : \
      (x <= 128) ? 7 : \
      (x <= 256) ? 8 : \
      (x <= 512) ? 9 : \
```

```
      (x <= 1024) ? 10 : \
      -1

// Home core hash function needs to respect 64 byte L1 cache size,
// or incorrect behavior results.
`define HOME_CORE(x) ((x >> `L1_LINE_SIZE_LOG_2) % `NUM_CORES)


`endif
```

## B.2 ring_cache.v

```
`include "defines.v"
`include "memory.v"
`include "signal_buffer.v"
`include "bundleizer.v"
`include "stopper.v"
`include "buffer.v"
`include "load_unit.v"

/*
 Assume that ring cache inputs from the core and L1 are outputs of registers from outside
 the module.
 */

module ring_cache(
                  input wire                                      clk,
                  input wire                                      reset,
                  input wire                                      flush,
                  input wire [`CORE_ID_WIDTH-1:0]                 firstIterationCoreId,

                  // Inputs and outputs to the owner core
                  input wire                                      coreCommandValid,
                  input wire [`TYPE_WIDTH-1:0]                    coreCommandType,
                  input wire [`ID_WIDTH-1:0]                      coreCommandId,
                  input wire [`ADDR_WIDTH-1:0]                    coreCommandAddr,
                  input wire [`DATA_WIDTH-1:0]                    coreCommandData,

                  output wire                                     coreCommandProcessed,
                  output wire [`DATA_WIDTH-1:0]                   coreResult,
                  output wire                                     coreLoadIsHit,

                  // Inputs and outputs from/to left neighbor
                  input wire                                      leftForwardBundleValid,
                  input wire [`FORWARD_NETWORK_BUNDLE_WIDTH-1:0]  leftForwardArrivingBundle,
                  output wire                                     leftForwardOutgoingCredit,

                  input wire                                      leftRequestBundleValid,
                  input wire [`REQUEST_NETWORK_BUNDLE_WIDTH-1:0]  leftRequestArrivingBundle,
                  output wire                                     leftRequestOutgoingCredit,

                  input wire                                      leftReplyBundleValid,
                  input wire [`REPLY_NETWORK_BUNDLE_WIDTH-1:0]    leftReplyArrivingBundle,
                  output wire                                     leftReplyOutgoingCredit,

                  // Inputs and outputs from/to right neighbor
                  input wire                                      rightForwardIncomingCredit,
```

```
                output wire                                     rightForwardBundleValid,
                output wire [`FORWARD_NETWORK_BUNDLE_WIDTH-1:0] rightForwardDepartingBundle,

                input wire                                      rightRequestIncomingCredit,
                output wire                                     rightRequestBundleValid,
                output wire [`REQUEST_NETWORK_BUNDLE_WIDTH-1:0] rightRequestDepartingBundle,

                input wire                                      rightReplyIncomingCredit,
                output wire                                     rightReplyBundleValid,
                output wire [`REPLY_NETWORK_BUNDLE_WIDTH-1:0]   rightReplyDepartingBundle,

                // Inputs and outputs from/to L1 cache
                input wire                                      writebackAccepted,
                input wire                                      writebackComplete,
                output wire                                     writebackValid,
                output wire [`ADDR_WIDTH-1:0]                   writebackAddr,
                output wire [`ADDR_WIDTH-1:0]                   writebackData,

                // Ports for read interaction with L1 cache
                input wire                                      cacheLoadAccepted,
                input wire                                      cacheLoadComplete,
                input wire [`DATA_WIDTH-1:0]                    cacheLoadResult,

                output wire                                     cacheLoadValid,
                output wire [`ADDR_WIDTH-1:0]                   cacheLoadAddr

);

    parameter CORE_ID = 0;

    // Combinational inputs to memory
    wire [`ADDR_WIDTH-1:0] addr_r;
    wire                   input_valid_r;

    wire [`ADDR_WIDTH-1:0] addressStore;
    wire [`DATA_WIDTH-1:0] dataStore;
    wire                   inputValidStore;

    // Outputs from memory
    wire                   memoryReadReady;
    wire                   memoryWriteReady;

    wire [`DATA_WIDTH-1:0] dataOutLoad;
    wire                   requestCompleteLoad;
    wire                   requestHitLoad;

    wire                   requestCompleteStore;
```

```
// Combinational outputs signal buffer, memory
wire                    coreWaitReleased;
wire                    coreWaitReleasedToStartFlush;
wire                    memoryFinishedFlush;


// Inputs to ReceiveBuffers. Releases the current oldest buffer/entry/bundle.
wire                    leftForwardReleaseBundle;
wire                    leftRequestReleaseBundle;
wire                    leftReplyReleaseBundle;


// Outputs from forwardReceiveBuffer
wire [`FORWARD_NETWORK_BUNDLE_WIDTH-1:0] leftForwardDepartingBundle;
wire                                     leftForwardValidDepartingBundle;
wire [`FORWARD_NETWORK_BUNDLE_WIDTH-1:0] requestBufferPeekA;
wire [`FORWARD_NETWORK_BUNDLE_WIDTH-1:0] requestBufferPeekB;

// Outputs from requestReceiveBuffer
wire [`REQUEST_NETWORK_BUNDLE_WIDTH-1:0] leftRequestDepartingBundle;
wire                                     leftRequestValidDepartingBundle;

// Outputs from replyReceiveBuffer
wire [`REPLY_NETWORK_BUNDLE_WIDTH-1:0]   leftReplyDepartingBundle;
wire                                     leftReplyValidDepartingBundle;

// Outputs from bundelizer and stopper
wire                                     coreInputServicedByBundle;
wire [`FORWARD_NETWORK_BUNDLE_WIDTH-1:0]  updatedBundle;
wire                                     updatedBundleValid;

wire [`FORWARD_NETWORK_BUNDLE_WIDTH-1:0]  prunedBundle;
wire                                     prunedBundleValid;

// Breakouts from updated bundle, to perform store to memory, and signal to signal buffer
wire [`ADDR_WIDTH-1:0]                   updatedStoreAddress;
wire [`DATA_WIDTH-1:0]                   updatedStoreData;
wire [`CORE_ID_WIDTH-1:0]                updatedStoreSenderCore;
wire                                     updatedStoreValid;

wire [(`SIGNAL_BANDWIDTH*`SIGNAL_ENTRY_WIDTH)-1:0] updatedSignals;

// Input to load unit, don't let request network pass forward network
wire                                             anyAddrMatch;

// Outputs from load unit, send to either: memory, core, request network, or reply network
```

```
wire [`ADDR_WIDTH-1:0]                          addressToLoad;
wire                                            addressToLoadValid;

wire                                            coreLoadProcessed;
wire [`DATA_WIDTH-1:0]                          coreLoadResult;
wire                                            coreLoadHit;

wire                                            rightRequestValid;
wire [`REQUEST_NETWORK_BUNDLE_WIDTH-1:0]        rightRequestDeparting;

wire                                            rightReplyValid;
wire [`REPLY_NETWORK_BUNDLE_WIDTH-1:0]          rightReplyDeparting;


// Outbound links and their credits
wire                                outboundForwardLinkReady;
reg [1:0]                           outboundForwardLinkCredits;

wire                                outboundRequestLinkReady;
reg [1:0]                           outboundRequestLinkCredits;

wire                                outboundReplyLinkReady;
reg [1:0]                           outboundReplyLinkCredits;


// Buffers for the three networks, all have the same basic interface.
// Arriving items from the coresponding links are stored in the buffer.
// Existing items are released if they are consumed by the ring cache node.
// A credit is sent to the ring cache node on the left if a buffer is consumed,
// to inform it of this fact.
assign outboundReplyLinkReady = (outboundReplyLinkCredits == 2'b00) ? 1'b0 : 1'b1;
receive_buffer #(.CORE_ID(CORE_ID), .ENTRY_WIDTH(`REPLY_NETWORK_BUNDLE_WIDTH))
          replyReceiveBuffer (
                              .reset(reset | flush),
                              .clk(clk),
                              .inputValid(leftReplyBundleValid),
                              .arrivingEntry(leftReplyArrivingBundle),
                              .outgoingCredit(leftReplyOutgoingCredit),
                              .releaseEntry(leftReplyReleaseBundle),
                              .departingEntry(leftReplyDepartingBundle),
                              .validDepartingEntry(leftReplyValidDepartingBundle)
                              );


assign outboundRequestLinkReady = (outboundRequestLinkCredits == 2'b00) ? 1'b0 : 1'b1;
receive_buffer #(.CORE_ID(CORE_ID), .ENTRY_WIDTH(`REQUEST_NETWORK_BUNDLE_WIDTH))
          requestReceiveBuffer (
```

```
                        .reset(reset | flush),
                        .clk(clk),
                        .inputValid(leftRequestBundleValid),
                        .arrivingEntry(leftRequestArrivingBundle),
                        .outgoingCredit(leftRequestOutgoingCredit),
                        .releaseEntry(leftRequestReleaseBundle),
                        .departingEntry(leftRequestDepartingBundle),
                        .validDepartingEntry(leftRequestValidDepartingBundle)
                        );



assign outboundForwardLinkReady = (outboundForwardLinkCredits == 2'b00) ? 1'b0 : 1'b1;
receive_buffer #(.CORE_ID(CORE_ID), .ENTRY_WIDTH(`FORWARD_NETWORK_BUNDLE_WIDTH))
          forwardReceiveBuffer (
                        .reset(reset | flush),
                        .clk(clk),
                        .inputValid(leftForwardBundleValid),
                        .arrivingEntry(leftForwardArrivingBundle),
                        .outgoingCredit(leftForwardOutgoingCredit),
                        .releaseEntry(leftForwardReleaseBundle),
                        .departingEntry(leftForwardDepartingBundle),
                        .validDepartingEntry(leftForwardValidDepartingBundle),

                        .peekA(requestBufferPeekA),
                        .peekB(requestBufferPeekB)

                        );

bundleizer #(.CORE_ID(CORE_ID))
             bundleLogic (
                              // Inputs from core
                              .coreCommandAddr(coreCommandAddr),
                              .coreCommandData(coreCommandData),
                              .coreCommandType(coreCommandType),
                              .coreCommandValid(coreCommandValid),
                              .coreCommandId(coreCommandId),

                              // Status of memory and link, needed to determine whether to release a buffer
                              .memoryReady(memoryWriteReady),
                              .outboundLinkReady(outboundForwardLinkReady),

                              // Outputs, inform core and forward network that something was consumed
                              .coreInputServiced(coreInputServicedByBundle),
                              .leftReleaseBundle(leftForwardReleaseBundle),

                              // Inputs from buffer
```

```
                                .leftDepartingBundle(leftForwardDepartingBundle),
                                .leftValidDepartingBundle(leftForwardValidDepartingBundle),

                                // Outputs to stopper
                                .outputBundle(updatedBundle),
                                .outputValid(updatedBundleValid)
                                );

// Process bundle coming from the bundleizer, pass any valid store to the memory,
// any valid signals to the signal buffer

// Valid bit for store
assign updatedStoreValid        = updatedBundleValid & updatedBundle[`STORE_ENTRY_WIDTH-1];

assign updatedStoreSenderCore   = updatedStoreValid == 1'b1 ?
                                  updatedBundle[`STORE_ENTRY_WIDTH-2:`ADDR_WIDTH+`DATA_WIDTH] :
                                  {`CORE_ID_WIDTH{1'b1}};

assign updatedStoreAddress      = updatedStoreValid == 1'b1 ?
                                  updatedBundle[`STORE_ENTRY_WIDTH-2-`CORE_ID_WIDTH:`DATA_WIDTH] :
                                  {`ADDR_WIDTH{1'b0}};

assign updatedStoreData         = updatedStoreValid == 1'b1 ?
                                  updatedBundle[`DATA_WIDTH-1:0] :
                                  {`DATA_WIDTH{1'b0}};

assign updatedSignals           = updatedBundleValid == 1'b1 ?
                                  updatedBundle[`FORWARD_NETWORK_BUNDLE_WIDTH-1:`STORE_ENTRY_WIDTH] :
                                  {((`SIGNAL_BANDWIDTH*`SIGNAL_ENTRY_WIDTH)){1'b0}};

// Handle writes to the memory, could be core or left receive buffer. Bundleizer has already chosen.
assign addressStore = updatedStoreValid ? updatedStoreAddress : {`ADDR_WIDTH{1'b0}};
assign dataStore = updatedStoreValid ? updatedStoreData : {`DATA_WIDTH{1'b0}};
assign inputValidStore = updatedStoreValid;


// Remove any stores/signals from the bundle that have reached their final core.  This happens
// after they are sent to memory/signal buffer, but before placed on the link.
stopper #(.CORE_ID(CORE_ID))
                    endLogic  (
                                .inputValid(updatedBundleValid),
                                .inputBundle(updatedBundle),
                                .outputBundle(prunedBundle),
                                .outputValid(prunedBundleValid)
                                );
```

```
memory #(.CORE_ID(CORE_ID)) ringCacheMemory (
                                .reset(reset | flush),
                                .clk(clk),

                                 // Outputs indicating load/store readiness
                                .readReady(memoryReadReady),
                                .writeReady(memoryWriteReady),

                                // Input load address from load unit
                                // (ultimately from core or request network)
                                .inputValidLoad(input_valid_r),
                                .addressLoad(addr_r),

                                // Outputs, sent to load unit
                                .dataOutLoad(dataOutLoad),
                                .requestCompleteLoad(requestCompleteLoad),
                                .requestHitLoad(requestHitLoad),

                                // Input store address/data from output of bundelizer
                                .inputValidStore(inputValidStore),
                                .addressStore(addressStore),
                                .dataStore(dataStore),

                                // Output
                                .requestCompleteStore(requestCompleteStore),

                                // Flush releated input and output, triggering start and indicating finish
                                .startFlush(coreWaitReleasedToStartFlush),
                                .finishedFlush(memoryFinishedFlush),

                                // Input and outputs to L1 cache
                                .writebackValid(writebackValid),
                                .writebackAddr(writebackAddr),
                                .writebackData(writebackData),
                                .writebackAccepted(writebackAccepted),
                                .writebackComplete(writebackComplete),

                                .cacheLoadAccepted(cacheLoadAccepted),
                                .cacheLoadComplete(cacheLoadComplete),
                                .cacheLoadResult(cacheLoadResult),
                                .cacheLoadValid(cacheLoadValid),
                                .cacheLoadAddr(cacheLoadAddr)
                                );

signal_buffer #(.EPOCH_BOUND(2), .RECEIVER_CORE(CORE_ID))
          signal_buffer (
                          .clk(clk),
```

```
                .reset(reset | flush),
                .firstIterationCoreId(firstIterationCoreId),

                // Signals from the bundelizer, send to signal buffer to record
                .incomingSignals(updatedSignals),

                // Incoming wait instruction from the core, output whether it is released or not.
                .incomingWaitValid(coreCommandValid == 1'b1 && coreCommandType == `TYPE_WAIT),
                .incomingWaitLight(coreCommandData[0]),
                .incomingWaitId(coreCommandId),

                .waitReleased(coreWaitReleased),
                .waitReleasedToStartFlush(coreWaitReleasedToStartFlush)
                );


load_unit #(.CORE_ID(CORE_ID))
        load_unit (
                .clk(clk),
                .reset(reset),

                // Incoming core command
                .coreCommandAddr(coreCommandAddr),
                .coreCommandType(coreCommandType),
                .coreCommandValid(coreCommandValid),

                // Incoming item on the request network
                .outboundRequestLinkReady(outboundRequestLinkReady),
                .leftRequestDepartingBundle(leftRequestDepartingBundle),
                .leftRequestValidDepartingBundle(leftRequestValidDepartingBundle),

                // Incoming item on the reply network
                .outboundReplyLinkReady(outboundReplyLinkReady),
                .leftReplyDepartingBundle(leftReplyDepartingBundle),
                .leftReplyValidDepartingBundle(leftReplyValidDepartingBundle),

                // Outputs from the memory - read hit status, and whether it is finished
                .requestCompleteLoad(requestCompleteLoad),
                .requestHitLoad(requestHitLoad),
                .dataOutLoad(dataOutLoad),

                // Check items in forwarding network buffers to make sure none
                // match anything in the request network buffers
                .peekA(requestBufferPeekA),
                .peekB(requestBufferPeekB),

                // Outputs from load unit, send to memory
```

```
                .addressToLoadValid(addressToLoadValid),
                .addressToLoad(addressToLoad),

                // Outputs from load unit, use to respond to core input
                .coreLoadProcessed(coreLoadProcessed),
                .coreLoadResult(coreLoadResult),
                .coreLoadHit(coreLoadHit),

                // Outputs from load unit, use to release a buffer from the request network,
                // and/or send a new one on the outgoing link
                .leftRequestReleaseBundle(leftRequestReleaseBundle),
                .rightRequestValid(rightRequestValid),
                .rightRequestDeparting(rightRequestDeparting),

                // Outputs from load unit, use to release a buffer from the reply network,
                // and/or send a new one on the outgoing link
                .leftReplyReleaseBundle(leftReplyReleaseBundle),
                .rightReplyValid(rightReplyValid),
                .rightReplyDeparting(rightReplyDeparting)
               );


// Connect load unit module to memory.  Not really necessary, but makes it more explicit.
assign input_valid_r = addressToLoadValid;
assign addr_r = addressToLoadValid ? addressToLoad : {`ADDR_WIDTH{1'b0}};


/********
 Outputs
********/

// Output pruned forward bundles (stores and signals) onto outgoing link
assign rightForwardBundleValid = prunedBundleValid;
assign rightForwardDepartingBundle = prunedBundle;

// Output proper request bundle (either from core or already in network, as decided by load unit)
assign rightRequestBundleValid = rightRequestValid;
assign rightRequestDepartingBundle = rightRequestDeparting;

// Output proper reply bundle (either from core or already in network, as decided by load unit)
assign rightReplyBundleValid = rightReplyValid;
assign rightReplyDepartingBundle = rightReplyDeparting;

// Return a hit if a load hit locally, or when returning from reply network.
// Currently, from the core's point of view, ALL loads are (eventually) hits.
// Load unit calculates this valuex
assign coreLoadIsHit = coreLoadHit;
```

```
    // Result is from local load or reply network.  Output from load unit.
    assign coreResult = coreLoadResult;

    // Core command completed if a local load hit, a store was injected to the forwarding network,
    // a normal wait was released, the special flush wait was released,
    // or a remote load initiated by this core came back on the reply network.
    assign coreCommandProcessed = coreLoadProcessed | coreInputServicedByBundle |
                                  coreWaitReleased | (memoryFinishedFlush & coreWaitReleasedToStartFlush);


    always@(posedge clk) begin
        if(reset == 1'b0) begin
            // New credit total = old credits + possible incoming credit, - something sent
            outboundForwardLinkCredits  <= outboundForwardLinkCredits + rightForwardIncomingCredit -
                                           (rightForwardBundleValid ? 1'b1 : 1'b0);

            outboundRequestLinkCredits <= outboundRequestLinkCredits + rightRequestIncomingCredit -
                                          (rightRequestBundleValid ? 1'b1 : 1'b0);

            outboundReplyLinkCredits   <= outboundReplyLinkCredits + rightReplyIncomingCredit -
                                          (rightReplyBundleValid ? 1'b1 : 1'b0);
        end
        else begin
            outboundForwardLinkCredits <= 2'd2;
            outboundRequestLinkCredits <= 2'd2;
            outboundReplyLinkCredits <= 2'd2;
        end
    end

endmodule
```

## B.3 BUFFER.V

```
`include "defines.v"


// Generic buffer used for three different pseduo-networks.
// Contains two slots for holding "bundles"/packets/elements.

module receive_buffer #(parameter ENTRY_WIDTH = (`FORWARD_NETWORK_BUNDLE_WIDTH), CORE_ID = 0) (
            input wire                      clk,
            input wire                      reset,

            // Inputs from left ring node.
            // If valid, we are positive that there is an empty buffer slot,
            // otherwise it would never have been sent.
            input wire                      inputValid,
            input wire [ENTRY_WIDTH-1:0]    arrivingEntry,

            // Outputs to left ring node.
            // Raise this to high for one cycle to communicate to the left node
            // that one of the buffers has become free.
            output wire                     outgoingCredit,

            // Inputs from local ring node.
            // This controls whether a waiting entry can be freed.
            input wire                      releaseEntry,

            // Outputs to local ring node.
            // If a valid departing entry, the ring cache might consume it by raising
            // the releaseEntry wire, informing the buffer to free it internally.
            output wire [ENTRY_WIDTH-1:0]   departingEntry,
            output wire                     validDepartingEntry,

            // Peek at buffers for address checking.
            // Used only by the request network to peek at the forwarding network, to make sure a remote load
            // isn't passing by a store to the same address.
            output wire [ENTRY_WIDTH-1:0]   peekA,
            output wire [ENTRY_WIDTH-1:0]   peekB
);

   // Two slots in the buffer
   reg [ENTRY_WIDTH-1:0]    entryA;
   reg [ENTRY_WIDTH-1:0]    entryB;

   // Valid bits not necessary depending on the data format of the entry, but we use them anyway.
   reg                      validA;
   reg                      validB;
```

```
reg                     current; // 0 means entryA is next to depart, 1 is B


// Combinational next values
reg [ENTRY_WIDTH-1:0] nextEntryA;
reg [ENTRY_WIDTH-1:0] nextEntryB;
reg                   nextValidA;
reg                   nextValidB;
reg                   nextCurrent;


// Reset and next state logic
always @(posedge clk) begin
   if(reset == 1'b1) begin
      entryA <= {ENTRY_WIDTH{1'b0}};
      entryB <= {ENTRY_WIDTH{1'b0}};
      validA <= 1'b0;
      validB <= 1'b0;
      current <= 1'b0;
   end
   else begin
      entryA <= nextEntryA;
      entryB <= nextEntryB;
      validA <= nextValidA;
      validB <= nextValidB;
      current <= nextCurrent;
   end

end

// Assign the peek outputs to the two buffers.
assign peekA = entryA;
assign peekB = entryB;

// Combinational logic for output of an entry.
// Check the "current" reg to determine which of the two buffers refers to
// the oldest valid entry.
assign validDepartingEntry = ((current == 1'b0) && (validA == 1'b1)) ||
                             ((current == 1'b1) && (validB == 1'b1)) ? 1'b1 : 1'b0;
assign departingEntry = ((current == 1'b0) && (validA == 1'b1)) ? entryA :
                        ((current == 1'b1) && (validB == 1'b1)) ? entryB :
                        {ENTRY_WIDTH{1'b0}};

// Release a credit to the left node if we have confirmed from the local node that
// a buffer is being consumed this cycle.
// The decision as to whether a buffer is being consumed is made early in the cycle,
```

```
// so we assume this signal can traverse to the adjacent core within a cycle.
// If this isn't realistic, it will need to be latched here first.
assign outgoingCredit = validDepartingEntry & releaseEntry ? 1'b1 : 1'b0;


// Muxes for next state values.
// Mux inputs are whether an entry is leaving the buffer, and whether an entry is arriving.
// 00: Nothing is changing, keep old values.
// 01: Nothing is leaving buffer, but something is arriving, so find proper slot for it.
// 10: Something is leaving, but not arriving.  Make sure to output the oldest valid entry.
// 11: Something is arriving and leaving.  Make sure to updated all registers properly.

// In general — when something is arriving, it goes into A if it is available, B otherwise.
// When something is leaving, it leaves first from A if current == 0, B if current == 1.
// When something is both leaving and arriving, the same rules apply.  It is impossible for
// something to be arriving if both buffers are alread full.

wire validEntryBeingReleased;
assign validEntryBeingReleased = validDepartingEntry & releaseEntry;

// Mux for nextEntryA
always@(*) begin
   case( {validEntryBeingReleased, inputValid} )
     2'b00: nextEntryA = entryA;
     2'b01: nextEntryA = ~(validA | validB) ? arrivingEntry :
                         validA ? entryA :
                         validB ? arrivingEntry :
                         {ENTRY_WIDTH{1'b0}};
     2'b10: nextEntryA = current ? entryA : {ENTRY_WIDTH{1'b0}};
     2'b11: nextEntryA = current ? arrivingEntry : {ENTRY_WIDTH{1'b0}};
   endcase
end

// Mux for nextValidA
always@(*) begin
   case( {validEntryBeingReleased, inputValid} )
     2'b00: nextValidA = validA;
     2'b01: nextValidA = 1'b1; // if something arriving, A always selected first
     2'b10: nextValidA = current ? validA : 1'b0;
     2'b11: nextValidA = current;
   endcase
end


// Mux for entryB.  Similar to A, but switching on inverse of 'current' reg
always@(*) begin
   case( {validEntryBeingReleased, inputValid} )
```

```
            2'b00: nextEntryB = entryB;
            2'b01: nextEntryB = ~(validA | validB) ? entryB :
                                validA ? arrivingEntry :
                                validB ? entryB :
                                {ENTRY_WIDTH{1'b0}};
            2'b10: nextEntryB = current ? {ENTRY_WIDTH{1'b0}} : entryB;
            2'b11: nextEntryB = current ? {ENTRY_WIDTH{1'b0}} : arrivingEntry;
        endcase
    end

    // Mux for nextValidB
    always@(*) begin
        case( {validEntryBeingReleased, inputValid} )
            2'b00: nextValidB = validB;
            2'b01: nextValidB = validA ? 1'b1 : validB;
            2'b10: nextValidB = current ? 1'b0 : validB;
            2'b11: nextValidB = ~current;
        endcase
    end

    // Mux for nextCurrent
    always@(*) begin
        case( {validEntryBeingReleased, inputValid} )
            2'b00: nextCurrent = current;
            2'b01: nextCurrent = ~(validA | validB) ? 1'b0 : current; // If nothing valid, A gets selected
            2'b10: nextCurrent = ~current; // Something leaving, toggle.
            2'b11: nextCurrent = ~current;
        endcase
    end

endmodule
```

## B.4 BUNDLEIZER.V

```
`include "defines.v"

// This module arbitrates between items already in the forwarding network
// (coming from the receive buffer module), and stores/signals injected by
// the core attached to this node.  Signals/stores circulate in the forwarding
// network together in a bundle.  A bundle consists of one or more stores and one or more
// signals.  Once items are in a bundle together, they must stay in the bundle together,
// until they have reached their final destination.  This is for corectness, as HELIX
// assumes that items injected to the ring cache are processed in-order.  If a newer signal
// passed an older store, a core could be erroronously unblocked before the correct data has arrived.

// A core can only add stores/signals to the bundle if there is an appropriate empty slot (store or signal)
// - otherwise it must wait for a bundle with an empty slot.
// If forwarding is blocked for any reason (pending eviction in the RC memory),
// then the entire bundle, stores and signals included must wait.

// Once a bundle passes through this module, it is sent to the RC memory and signal buffer for processing.
// It is only sent to the output ports if it can proceed to the memory and outgoing link this cycle.
// The signal buffer can not block, so is always ready for inputs.

// A bundle has a fixed number of slots for signals, corresponding to the available signal bandwidth in the
// signal buffer, and a fixed number of slots for stores, corresponding to the available data bandwidth in the
// ring cache memory.

// There are four different scenarios to handle:

// Nothing leaving receive buffer, no core store/signal being injected
// Bundle leaving receive buffer, no core store/signal being injected, so pass it on unchanged
// Nothing leaving receive buffer, core store/signal being injected,
//                  so create new bundle with only core store/signal
// Bundle leaving receive buffer, core store/signal being injected, add it only if a free spot


module bundleizer(
                    // Inputs from core, could be either a store or a signal
                    input wire                                  coreCommandValid,
                    input wire [`TYPE_WIDTH-1:0]                coreCommandType,
                    input wire [`ID_WIDTH-1:0]                  coreCommandId,
                    input [`ADDR_WIDTH-1:0]                     coreCommandAddr,
                    input [`DATA_WIDTH-1:0]                     coreCommandData,

                    // Inputs from forwarding network recieve buffer, indicating if it
                    // holds a valid bundle, and if so, its contents.
                    input wire                                  leftValidDepartingBundle,
                    input wire [`FORWARD_NETWORK_BUNDLE_WIDTH-1:0]  leftDepartingBundle,
```

```
                // Readiness inputs from the RC memory and link to the adjacent RC node.
                input wire                                      memoryReady,
                input wire                                      outboundLinkReady,

                // Output to the receive buffer to tell it that it can release the current bundle
                output wire                                     leftReleaseBundle,

                // The updated bundle being sent to the RC memory, signal buffer, and outgoing link
                output wire [`FORWARD_NETWORK_BUNDLE_WIDTH-1:0] outputBundle,
                output wire                                     outputValid,

                // An output to tell the core that its new item has been added to a bundle
                output wire                                     coreInputServiced
);

    parameter CORE_ID = 0;

    // Helper control wires, indicating whether the incoming bundle / core inputs are present.
    wire coreHasStore;
    wire coreHasSignal;
    wire leftHasStore;
    wire leftHasSignal;

    // Wires controlling which of the inputs we will pass on to the output bundle.
    wire serviceCoreSignal;
    wire serviceCoreStore;
    wire serviceLeftStore;

    wire [`SIGNAL_BANDWIDTH-1:0] leftValidSignals;  // Which signal slots are already taken, one bit per signal.
    wire                         leftFullSignals;   // Is the left bundle entirely full.
    wire [`SIGNAL_BANDWIDTH-1:0] serviceLeftSignal; // Which signals from incoming bundle that we will pass on.

    // Determine which signals from left bundle are valid, in a signal bandwidth agnostic away.
    genvar                       i;
    generate
        for (i=0; i < `SIGNAL_BANDWIDTH; i=i+1) begin : validSignals
            assign leftValidSignals[i] = leftValidDepartingBundle &
                                         (leftDepartingBundle[(`SIGNAL_ENTRY_WIDTH * (i+1)) + `STORE_ENTRY_WIDTH - 1]);
        end
    endgenerate

    // Valid bit for store in incoming bundle.
    assign leftHasStore  = leftValidDepartingBundle & leftDepartingBundle[`STORE_ENTRY_WIDTH-1];

    // OR all signal valid bits to determine if any signals are valid.
    assign leftHasSignal = | leftValidSignals;
```

```
// AND all signal bits to determine if any empty space in bundle for a new signal.
assign leftFullSignals = & leftValidSignals;

assign coreHasStore  = (coreCommandValid == 1'b1 && coreCommandType == `TYPE_STORE) ? 1'b1 : 1'b0;
assign coreHasSignal = (coreCommandValid == 1'b1 && coreCommandType == `TYPE_SIGNAL) ? 1'b1 : 1'b0;

assign serviceCoreSignal = (coreHasSignal == 1'b1 && leftFullSignals == 1'b0) ? 1'b1 : 1'b0;
assign serviceCoreStore =  (coreHasStore  == 1'b1  && leftHasStore == 1'b0) ? 1'b1 : 1'b0;

assign serviceLeftSignal = leftValidSignals;   // All valid signals get servied before any core signal.
assign serviceLeftStore =  (leftHasStore == 1'b1) ? 1'b1 : 1'b0;

// Outputs from bundleizer, informing core and/or receive buffer that they can release
// their current input/bundle. Only set high if it is guarenteed the bundle will send
// (i.e. memory is ready and outbound link is ready).
assign coreInputServiced = (memoryReady == 1'b0 || outboundLinkReady == 1'b0)    ? 1'b0 :
                           serviceCoreSignal == 1'b1 || serviceCoreStore == 1'b1 ? 1'b1 :
                           1'b0;

// As long as the memory and link are ready, and left has any valid bundle, it can be sent.
// Output this so the receive buffer can release the bundle
assign leftReleaseBundle = (memoryReady == 1'b0 || outboundLinkReady == 1'b0) ? 1'b0 :
                          (leftHasSignal == 1'b1 || leftHasStore == 1'b1) ? 1'b1 :
                          1'b0;


// Use above calculated values to select proper stores/signals to send to outbound bundle
wire [`STORE_ENTRY_WIDTH-1:0]                          coreStore;
wire [`SIGNAL_ENTRY_WIDTH-1:0]                         coreSignal;

wire [`STORE_ENTRY_WIDTH-1:0]                          leftStore;
wire [`SIGNAL_ENTRY_WIDTH-1:0]                         leftSignal;

wire [`STORE_ENTRY_WIDTH-1:0]                          chosenStore;
wire [(`SIGNAL_BANDWIDTH * `SIGNAL_ENTRY_WIDTH)-1:0]   chosenSignals;

wire [`CORE_ID_WIDTH-1:0]                              coreId;

assign coreId = CORE_ID[`CORE_ID_WIDTH-1:0];

// Construct potential new entries for core/left store and signal.
// Core id is part of the entry so that the entry can be removed from the bundle
// when it has reached core # (coreId - 1 % NUM_CORES).
assign coreStore  = {1'b1, coreId, coreCommandAddr, coreCommandData};
assign coreSignal = {1'b1, coreId, coreCommandId};
```

```
    assign leftStore  = leftDepartingBundle[`STORE_ENTRY_WIDTH-1:0];

    // Find the rightmost 1 bit in emptySlots, use that to add new signal to bundle
    wire [`SIGNAL_BANDWIDTH-1:0] emptySlots = ~leftValidSignals;
    wire [`SIGNAL_BANDWIDTH-1:0] rightmostEmptySlot = emptySlots & (-emptySlots);

    // Assign signals to output bundle in signal bandwidth agnostic way.
    // For the ith signal entry in bundle, check if it is the selected slot for a new core signal.
    // Otherwise, check to make sure the signal in the bundle is valid, and if so, pass it on to the output.
    generate
       for(i = 0; i < `SIGNAL_BANDWIDTH; i = i + 1) begin : chooseSignals
          assign chosenSignals[(`SIGNAL_ENTRY_WIDTH * (i+1))-1:(`SIGNAL_ENTRY_WIDTH * (i))] =
                      (serviceCoreSignal == 1'b1 && rightmostEmptySlot[i] == 1'b1) ?
                      coreSignal :
                      serviceLeftSignal[i] == 1'b1 ?
                      leftDepartingBundle[(`SIGNAL_ENTRY_WIDTH * (i+1))+`STORE_ENTRY_WIDTH-1 :
                                          (`SIGNAL_ENTRY_WIDTH * (i))+`STORE_ENTRY_WIDTH] :
                      {`SIGNAL_ENTRY_WIDTH{1'b0}};
       end
    endgenerate

    // Choose which of left or core get to be sent to outgoing bundle
    assign chosenStore = serviceCoreStore == 1'b1 ? coreStore :
                         serviceLeftStore == 1'b1 ? leftStore :
                         {`STORE_ENTRY_WIDTH{1'b0}};


    // Output is only valid if both the memory and link are available, and there is something in the bundle
    assign outputValid = (coreInputServiced == 1'b1 || leftReleaseBundle == 1'b1) ? 1'b1 : 1'b0;

    // Output selected signals and stores as the new bundle
    assign outputBundle = (outputValid == 1'b1) ?
                          {chosenSignals, chosenStore} :
                          {`FORWARD_NETWORK_BUNDLE_WIDTH{1'b0}};

endmodule
```

## B.5 STOPPER.V

```
`include "defines.v"

// This module removes stores and signals that are at their terminal location from the circulating bundle.

module stopper(
               // The input coresponds to the output from the bundleizer module.
               // These items have already been sent to the RC memory and signal buffer,
               // and have been cleared to access the outgoing link if appropriate.
                  input wire                                    inputValid,
                  input wire [`FORWARD_NETWORK_BUNDLE_WIDTH-1:0]  inputBundle,

               // Any items not at their final destination (i.e., haven't traversed the entire ring yet)
               // are allowed to pass on to the outgoing link.
                  output wire [`FORWARD_NETWORK_BUNDLE_WIDTH-1:0] outputBundle,
                  output wire                                   outputValid
);

   parameter CORE_ID = 0;

   wire [`CORE_ID_WIDTH-1:0]                                    coreId;

   assign coreId = CORE_ID[`CORE_ID_WIDTH-1:0];

   // Only pass store along if this isn't its final desitination.  Any stores that have traversed the ring
   // are removed from the bundle.
   wire [`CORE_ID_WIDTH-1:0]                                    bundleStoreTerminatingCoreId;
   wire [`STORE_ENTRY_WIDTH-1:0]                                bundleStore;
   wire [`STORE_ENTRY_WIDTH-1:0]                                chosenStore;
   assign bundleStoreTerminatingCoreId = inputBundle[`STORE_ENTRY_WIDTH-2:`ADDR_WIDTH+`DATA_WIDTH] - 1'b1;
   assign bundleStore  = inputBundle[`STORE_ENTRY_WIDTH-1:0];
   assign chosenStore  = (bundleStoreTerminatingCoreId == coreId) ? {`STORE_ENTRY_WIDTH{1'b0}} : bundleStore;


   // Only pass signals along if this isn't its final desitination.  Slightly ugly due to need to handle
   // different signal bandwidths. Any signals that have traversed the ring
   // are removed from the bundle,
   wire [`CORE_ID_WIDTH-1:0]                                    bundleSignalTerminatingCoreId [0:`SIGNAL_BANDWIDTH-1];
   wire [`SIGNAL_ENTRY_WIDTH-1:0]                               bundleSignals [0:`SIGNAL_BANDWIDTH-1];
   wire [(`SIGNAL_BANDWIDTH * `SIGNAL_ENTRY_WIDTH)-1:0]         chosenSignals;
   genvar                                                       i;
   generate
    for (i=0; i < `SIGNAL_BANDWIDTH; i=i+1) begin : choseSignals
      assign bundleSignalTerminatingCoreId[i] = inputBundle[(`SIGNAL_ENTRY_WIDTH * (i+1))+`STORE_ENTRY_WIDTH -2:
                                                            (`SIGNAL_ENTRY_WIDTH * (i))+`ID_WIDTH+`STORE_ENTRY_WIDTH]-1'b1;
```

```
        assign bundleSignals[i] = inputBundle[(`SIGNAL_ENTRY_WIDTH * (i+1)) + `STORE_ENTRY_WIDTH - 1:
                                              (`SIGNAL_ENTRY_WIDTH * (i))+`STORE_ENTRY_WIDTH];

        assign chosenSignals[(`SIGNAL_ENTRY_WIDTH * (i+1))-1:(`SIGNAL_ENTRY_WIDTH * (i))] =
                                    (bundleSignalTerminatingCoreId[i] == coreId) ? {`SIGNAL_ENTRY_WIDTH{1'b0}} :
                                    bundleSignals[i];
    end
    endgenerate

    // Output is valid if input is valid, and the bundle has any remaining valid stores/signals
    assign outputValid = (inputValid == 1'b1) ? (|chosenStore) | (|chosenSignals) : 1'b0;
    assign outputBundle = {chosenSignals, chosenStore};

endmodule
```

## B.6 load_unit.v

```
`include "defines.v"

// This module handles all of the logic to handle loads in the ring cache.
// Loads have two sources - from the local core, or from a remote core.
// This unit arbitrates between who gets the load port of the memory,
// makes the request, and processes the result.

// If a core performs a load, and it misses, the result must be sent over the request network
// to find the home core for the address.
// Once a remote load, read from the request network, accesses the local memory, the reply
// is sent back over the reply network.  Once the reply returns to the originating core,
// this module outputs the loaded value back to the core.

module load_unit(
                    input wire                                      clk,
                    input wire                                      reset,

                    // Inputs and outputs to the owner core
                    input wire                                      coreCommandValid,
                    input wire [`TYPE_WIDTH-1:0]                    coreCommandType,
                    input wire [`ADDR_WIDTH-1:0]                    coreCommandAddr,

                    // Inputs from the request and reply networks
                    input wire                                      outboundRequestLinkReady,
                    input wire                                      leftRequestValidDepartingBundle,
                    input wire [`REQUEST_NETWORK_BUNDLE_WIDTH-1:0]  leftRequestDepartingBundle,

                    input wire                                      outboundReplyLinkReady,
                    input wire                                      leftReplyValidDepartingBundle,
                    input wire [`REPLY_NETWORK_BUNDLE_WIDTH-1:0]    leftReplyDepartingBundle,

                    // Result of the last memory load operation
                    input wire                                      requestCompleteLoad,
                    input wire                                      requestHitLoad,
                    input wire [`DATA_WIDTH-1:0]                    dataOutLoad,


                   // Peeks into the forwarding network receive buffers
                    input wire [`FORWARD_NETWORK_BUNDLE_WIDTH-1:0]   peekA,
                    input wire [`FORWARD_NETWORK_BUNDLE_WIDTH-1:0]   peekB,

                    // The selected address to next load from the memory
                    output wire                                     addressToLoadValid,
                    output wire [`ADDR_WIDTH-1:0]                   addressToLoad,
```

```
                // Outputs to the core with the result of its loads
                output wire                                     coreLoadProcessed,
                output wire [`DATA_WIDTH-1:0]                   coreLoadResult,
                output wire                                     coreLoadHit,

                // Outputs to send a new request/reply network item, and/or to release the current one
                output wire                                     leftRequestReleaseBundle,
                output wire                                     rightRequestValid,
                output wire [`REQUEST_NETWORK_BUNDLE_WIDTH-1:0] rightRequestDeparting,

                output wire                                     leftReplyReleaseBundle,
                output wire                                     rightReplyValid,
                output wire [`REPLY_NETWORK_BUNDLE_WIDTH-1:0]   rightReplyDeparting
);

    parameter CORE_ID = 0;

    // Track what the read port is doing
    wire                    memServicingRemoteLoad;
    wire                    memServicingCoreLoad;
    reg                     pendingCoreLoad;
    reg                     pendingRemoteLoad;

    // Indicator bit for core load waiting to enter request network
    reg                     coreLoadEnqueuingRequestNetwork;
    reg                     coreLoadWaitingForReply;

    // Indicator bit for remote load waiting to enter reply network
    reg                     remoteLoadEnqueuingReplyNetwork;

    // For remote loads accessing this ring cache memory
    reg                                     remoteLoadWaitingValid;
    reg [`REQUEST_NETWORK_BUNDLE_WIDTH-1:0] remoteLoadWaiting;

    // For remote loads waiting to access reply network
    reg [`DATA_WIDTH-1:0]   remoteLoadData;

    // Combinational values related to request network
    wire                    coreLoadEnqueuingRequestNetworkNext;
    wire                    remoteLoadEnqueuingReplyNetworkNext;
    wire                    requestNetCanProceed;
    wire                    releaseCoreLoadToRequestNetwork;
    wire                    leftRequestLeaveNetwork;
    wire [`ADDR_WIDTH-1:0]  leftRequestBundleHomeCore;
    wire                    leftRequestBundleFoundHome;
```

```
// Combinational values related to the reply network
wire                    leftReplyBundleReturnedToOrigin;
wire                    releaseRemoteLoadToReplyNetwork;


// Memory is servicing a load from the core if it already was, or if there's a new load waiting,
// and not a load in the request network waiting.
assign memServicingCoreLoad = pendingCoreLoad |
                              (coreCommandValid == 1'b1 && coreCommandType == `TYPE_LOAD &&
                               coreLoadEnqueuingRequestNetwork == 1'b0 && coreLoadWaitingForReply == 1'b0 &&
                               pendingRemoteLoad == 1'b0 && remoteLoadWaitingValid == 1'b0);

// Memory is servicing a load from the remote load network if it already was,
// or if there is a new remote load waiting, and not one occupying the send buffer.
assign memServicingRemoteLoad = pendingRemoteLoad |
                    (~pendingCoreLoad & remoteLoadWaitingValid & ~remoteLoadEnqueuingReplyNetwork);


/*****************
 Outputs to memory
***************/

// Memory input is valid if we are servicing any load.
assign addressToLoadValid = memServicingCoreLoad | memServicingRemoteLoad;

// Set the read address based on whether we are servicing the core or the request network.
assign addressToLoad = memServicingCoreLoad ? coreCommandAddr :
               memServicingRemoteLoad ?  remoteLoadWaiting[`ADDR_WIDTH-1:0] :
               {`ADDR_WIDTH{1'b0}};


/********************
 Request network logic
********************/

// Request network must not pass forwarding network if it contains any stores that match the
// address of the request being forwarded. It is very very unlikely that this would occur,
// but is necessary for correctness. Peek into receive buffer for forwarding network,
// compare to candidate for leaving the request network.
wire                                    peekAValid;
wire                                    peekBValid;
wire [`ADDR_WIDTH-1:0]                  peekAAddr;
wire [`ADDR_WIDTH-1:0]                  peekBAddr;
wire                                    peeksMatchA;
wire                                    peeksMatchB;
wire                                    anyAddrMatch;
```

```
assign peekAValid = peekA[`STORE_ENTRY_WIDTH-1];
assign peekBValid = peekB[`STORE_ENTRY_WIDTH-1];

assign peekAAddr = peekA[`DATA_WIDTH+`ADDR_WIDTH-1:`DATA_WIDTH];
assign peekBAddr = peekB[`DATA_WIDTH+`ADDR_WIDTH-1:`DATA_WIDTH];

assign peeksMatchA = peekAValid == 1'b1 && leftRequestValidDepartingBundle == 1'b1 &&
                     peekAAddr == leftRequestDepartingBundle[`ADDR_WIDTH-1:0] ? 1'b1 : 1'b0;

assign peeksMatchB = peekBValid == 1'b1 && leftRequestValidDepartingBundle == 1'b1 &&
                     peekBAddr == leftRequestDepartingBundle[`ADDR_WIDTH-1:0] ? 1'b1 : 1'b0;

assign anyAddrMatch = peeksMatchA | peeksMatchB;


assign requestNetCanProceed = (~anyAddrMatch);

// Check if arriving request needs to access this cores ring cache node
assign leftRequestBundleHomeCore = `HOME_CORE(leftRequestDepartingBundle[`ADDR_WIDTH-1:0]);
assign leftRequestBundleFoundHome = leftRequestBundleHomeCore[`CORE_ID_WIDTH-1:0] ==
                                    CORE_ID[`CORE_ID_WIDTH-1:0] ? 1'b1 : 1'b0;

// Check if it is valid for item in request network to exit and access local node
assign leftRequestLeaveNetwork = leftRequestValidDepartingBundle &
                                 leftRequestBundleFoundHome & ~remoteLoadWaitingValid;

// Core load can enter request network if there is one waiting, and the outgoing link is ready,
// and the request network doesn't have something to forward, or if it does,
// is its final destination this node.  Item in network always has priority
assign releaseCoreLoadToRequestNetwork = coreLoadEnqueuingRequestNetwork & outboundRequestLinkReady &
                                         (~leftRequestValidDepartingBundle |
                                            (leftRequestLeaveNetwork & requestNetCanProceed) );


/*************************
 Outputs to Request network
*************************/

// Release item from request network buffer if it is accessing this node,
// or if it is being passed on to next node on the right
assign leftRequestReleaseBundle = requestNetCanProceed &
                                  ( leftRequestLeaveNetwork |
                                    (leftRequestValidDepartingBundle & ~leftRequestBundleFoundHome &
                                     outboundRequestLinkReady
                                    )
                                  );
```

```
// Send request network item to the right if there is one to pass on,
// or a new one injected from the core.
assign rightRequestValid = (leftRequestReleaseBundle & (~leftRequestLeaveNetwork)) |
                           releaseCoreLoadToRequestNetwork;

assign rightRequestDeparting = leftRequestReleaseBundle & ~leftRequestLeaveNetwork  ?
                               leftRequestDepartingBundle :
                               releaseCoreLoadToRequestNetwork ?
                               {1'b1, CORE_ID[`CORE_ID_WIDTH-1:0], coreCommandAddr} :
                               {`REQUEST_NETWORK_BUNDLE_WIDTH{1'b0}};

/*******************
 Reply network logic
*******************/

// Check if the oldest item in the reply network buffer originated from this core
assign leftReplyBundleReturnedToOrigin = (leftReplyValidDepartingBundle) &
                    (leftReplyDepartingBundle[`REPLY_NETWORK_BUNDLE_WIDTH-2:`DATA_WIDTH] ==
                    CORE_ID[`CORE_ID_WIDTH-1:0] ? 1'b1 : 1'b0);

// If there is a remote load done processing, and there isn't an item in the reply network,
// release the remote load to the reply network.
// It can also be released if the current item in the reply network is exiting at this node.
assign releaseRemoteLoadToReplyNetwork = outboundReplyLinkReady & remoteLoadEnqueuingReplyNetwork &
                                         ( (leftReplyValidDepartingBundle & leftReplyBundleReturnedToOrigin)
                                           | (~leftReplyValidDepartingBundle)
                                         );



/************************
 Outputs to reply network
************************/

// Release item from reply network buffer if it is accessing this node,
// or if it is being passed on to next node on the right
assign leftReplyReleaseBundle = leftReplyValidDepartingBundle &
                    ( (~leftReplyBundleReturnedToOrigin & outboundReplyLinkReady) |
                      leftReplyBundleReturnedToOrigin);

// Send reply network item to the right if there is one to pass on,
// or a recently finished remote load waiting to enter the reply network.
assign rightReplyValid = (leftReplyReleaseBundle & (~leftReplyBundleReturnedToOrigin)) |
                         releaseRemoteLoadToReplyNetwork;

assign rightReplyDeparting = leftReplyReleaseBundle & ~leftReplyBundleReturnedToOrigin  ?
                    leftReplyDepartingBundle :
```

```
                    releaseRemoteLoadToReplyNetwork ?
                    {remoteLoadWaiting[`REQUEST_NETWORK_BUNDLE_WIDTH-1:`ADDR_WIDTH] , remoteLoadData} :
                    {`REPLY_NETWORK_BUNDLE_WIDTH{1'b0}};

/****************
 Outputs to core
***************/

assign coreLoadProcessed = (memServicingCoreLoad & requestCompleteLoad & requestHitLoad) |
                           leftReplyBundleReturnedToOrigin;

assign coreLoadHit = (memServicingCoreLoad & requestCompleteLoad & requestHitLoad) |
                     leftReplyBundleReturnedToOrigin;

assign coreLoadResult = (memServicingCoreLoad & requestCompleteLoad & requestHitLoad) ? dataOutLoad :
                    leftReplyBundleReturnedToOrigin ? leftReplyDepartingBundle[`DATA_WIDTH-1:0] :
                    {`DATA_WIDTH{1'b0}};


/*****************
 Next state values
*****************/

// Enqueue core load on request network if it missed locally
assign coreLoadEnqueuingRequestNetworkNext = memServicingCoreLoad & ~requestHitLoad & requestCompleteLoad;

// After remote load is complete, enqueue to reply network
assign remoteLoadEnqueuingReplyNetworkNext = memServicingRemoteLoad & requestCompleteLoad;


// State transitions
always@(posedge clk) begin
   if(reset == 1'b0) begin
      // Remote loads started from this core
      if(coreLoadEnqueuingRequestNetworkNext == 1'b1) begin
         coreLoadEnqueuingRequestNetwork <= 1'b1;
      end
      else if(releaseCoreLoadToRequestNetwork == 1'b1) begin
         coreLoadEnqueuingRequestNetwork <= 1'b0;
         coreLoadWaitingForReply <= 1'b1;
      end
      else if(leftReplyBundleReturnedToOrigin == 1'b1) begin
         coreLoadWaitingForReply <= 1'b0;
      end


      // Remote loads from other cores
```

```
            if(leftRequestLeaveNetwork == 1'b1) begin
               remoteLoadWaitingValid <= 1'b1;
               remoteLoadWaiting <= leftRequestDepartingBundle;
            end
            else if(remoteLoadEnqueuingReplyNetworkNext == 1'b1) begin
               remoteLoadEnqueuingReplyNetwork <= 1'b1;
               remoteLoadData <= dataOutLoad;
            end
            else if(releaseRemoteLoadToReplyNetwork == 1'b1) begin
               remoteLoadWaitingValid <= 1'b0;
               remoteLoadWaiting <= {`REQUEST_NETWORK_BUNDLE_WIDTH{1'b0}};
               remoteLoadEnqueuingReplyNetwork <= 1'b0;
               remoteLoadData <= {`DATA_WIDTH{1'b0}};
            end

            // State of read port.  If request didn't complete, record this fact.
            pendingCoreLoad <= memServicingCoreLoad & ~requestCompleteLoad;;
            pendingRemoteLoad <= memServicingRemoteLoad & ~requestCompleteLoad;
         end
         else begin
            coreLoadEnqueuingRequestNetwork <= 1'b0;
            coreLoadWaitingForReply <= 1'b0;

            remoteLoadWaitingValid <= 1'b0;
            remoteLoadWaiting <= {`REQUEST_NETWORK_BUNDLE_WIDTH{1'b0}};

            remoteLoadData <= {`DATA_WIDTH{1'b0}};

            remoteLoadEnqueuingReplyNetwork <= 1'b0;

            pendingCoreLoad <= 1'b0;
            pendingRemoteLoad <= 1'b0;
         end
      end

endmodule
```

## B.7 MEMORY.V

```
`include "defines.v"
`include "priority_encoder.v"
`include "bloom_filter.v"
`include "array.v"

/* This module wraps the array module, and processes loads and stores for the ring cache.
 */

`define STATE_FLUSHING 2'd0
`define STATE_PENDING_EVICTION 2'd1
`define STATE_READY 2'd2

`define STATE_L1_LOAD 2'd1

module memory(
                    input wire                      reset,
                    input wire                      clk,
                    output wire                     readReady,
                    output wire                     writeReady,

                    // Inputs for loads, either from core or request network
                    input wire                      inputValidLoad,
                    input wire [`ADDR_WIDTH-1:0]    addressLoad,

                    // Combinational outputs
                    output wire [`DATA_WIDTH-1:0]   dataOutLoad,
                    output wire                     requestCompleteLoad,
                    output wire                     requestHitLoad,

                    // Inputs for stores
                    input wire                      inputValidStore,
                    input wire [`ADDR_WIDTH-1:0]    addressStore,
                    input wire [`DATA_WIDTH-1:0]    dataStore,

                    // Inputs/outputs for flush
                    input wire                      startFlush,
                    output wire                     finishedFlush,

                    // Combinational outputs
                    output wire                     requestCompleteStore,

                    // Ports for writeback interaction with L1 cache
                    input wire                      writebackAccepted,
                    input wire                      writebackComplete,
                    output wire                     writebackValid,
```

```
                output wire [`ADDR_WIDTH-1:0] writebackAddr,
                output wire [`ADDR_WIDTH-1:0] writebackData,

                // Ports for read interaction with L1 cache
                input wire                    cacheLoadAccepted,
                input wire                    cacheLoadComplete,
                input wire [`DATA_WIDTH-1:0]  cacheLoadResult,

                output wire                   cacheLoadValid,
                output wire [`ADDR_WIDTH-1:0] cacheLoadAddr


);

    parameter CORE_ID = 0;

    parameter NUM_ENTRIES = 256;
    parameter ASSOC = 1; // must be 1 for now

    localparam NUM_SETS = (NUM_ENTRIES / ASSOC);
    localparam NUM_INDEX_BITS = `CLOG2(NUM_SETS);
    localparam NUM_TAG_BITS   = `ADDR_WIDTH - 2 - NUM_INDEX_BITS;
    localparam WAY_ENTRY_SIZE =(ASSOC*(1+NUM_TAG_BITS+`DATA_WIDTH));


    // Combinational intermediate values dependent on request, state, etc
    wire                                initiateNewWriteback;

    // State bits
    reg [1:0]                           readState;
    reg [1:0]                           writeState;

    // Combinational next state
    reg [1:0]                           nextReadState;
    reg [1:0]                           nextWriteState;


    /*
     Eviction/flush related:
    */

    // Maximum number of items pending eviction <= number of entries in the data array, likely far, far fewer
    reg [NUM_INDEX_BITS-1:0]            evictionsAwaitingL1Confirmation;

    //
    reg                                 pendingEvictValid;
    reg [`DATA_WIDTH-1:0]               pendingEvictData;
```

```
reg [`ADDR_WIDTH-1:0]                   pendingEvictAddr;

// Flush related state and combinational logic
reg                                     flushWalkDone;
reg [NUM_SETS-1:0]                      ownerBitset ;
wire [NUM_SETS-1:0]                     nextIndexBitToFlush;
wire [NUM_INDEX_BITS-1:0]               nextIndexToFlush;


/*
 L1 load related:
*/

// State
reg                                     pendingL1LoadValid;
reg                                     pendingL1LoadAccepted;
reg [`ADDR_WIDTH-1:0]                   pendingL1LoadAddr;

// Track which values have already been seen, so uncessary request network accesses are avoided.
wire                                    hashTableMiss;


/*
 Memory array inputs/outputs
 */

wire                                    port1Hit;
wire [`DATA_WIDTH-1:0]                  port1DataOut;

wire                                    port2Valid;
wire                                    port2WriteEnable;
wire [`ADDR_WIDTH-1:0]                  port2Address;

wire                                    port2Eviction;
wire [`ADDR_WIDTH-1:0]                  port2ExistingAddr;
wire [`DATA_WIDTH-1:0]                  port2ExistingData;
wire                                    port2Hit;

wire [`DATA_WIDTH-1:0]                  port2DataOut;


/*
 Misc. memory address properties
*/

// For reads, to know if the L1 should be accessed instead of request network
wire [`ADDR_WIDTH-1:0]                  port1AddressHomeCore;
```

```
wire                                port1AddressHomeCoreMatches;

// For store evictions, to know whether to writeback the evicted value to the L1
wire [`ADDR_WIDTH-1:0]              port2ExistingAddressHomeCore;
wire                                port2ExistingAddressHomeCoreMatches;

// For stores, to record which stored values are 'owned' by this node
wire [`ADDR_WIDTH-1:0]              port2AddressHomeCore;
wire                                port2AddressHomeCoreMatches;


// The actual memory array and logic.
array #(.NUM_ENTRIES(NUM_ENTRIES), .ASSOC(ASSOC)) array
  (
   .clk(clk),
   .reset(reset),

   .port1Valid(inputValidLoad),
   .port1Address(addressLoad),
   .port1WriteData({`DATA_WIDTH{1'b0}}),
   .port1WriteEnable(1'b0),

   .port2Valid(port2Valid),
   .port2Address(port2Address),
   .port2WriteData(dataStore),
   .port2WriteEnable(port2WriteEnable),

   .port1DataOut(port1DataOut),
   .port1Hit(port1Hit),
   .port1Complete(),
   .port1Eviction(),
   .port1ExistingData(),
   .port1ExistingAddr(),

   .port2DataOut(port2DataOut),
   .port2Hit(port2Hit),
   .port2Complete(),
   .port2Eviction(port2Eviction),
   .port2ExistingData(port2ExistingData),
   .port2ExistingAddr(port2ExistingAddr)

   );


// Hashtable / bloom filter tracks which addresses have
// already been seen by this node.  If an address has
// never been seen before, and it is being loaded and misses,
```

```
// it can be loaded directly from the L1 even if this isn't the home core
// for that address.
bloom_filter hashLookup
(
 .addrToCheck(addressLoad),
 .addrToSet(addressStore),
 .addrToSetValid(inputValidStore & writeReady),
 .reset(reset),
 .clk(clk),

 .hashTableMiss(hashTableMiss)
);


// Flush helper.  Uses the ownerBitset to track which stored addresses that
// this node owns.  This makes the flush more efficient by only loading and writing back
// the proper entries in the memory array.  Currently assumes direct mapped cache.
assign nextIndexBitToFlush = ownerBitset;
priority_encoder #(.NUM_SETS(NUM_SETS), .NUM_INDEX_BITS(NUM_INDEX_BITS)) ownerEncoder
(
 .ownerBitset(ownerBitset),
 .nextIndexToFlush(nextIndexToFlush)
);



/***************
 Store port logic
****************/

wire [NUM_INDEX_BITS-1:0]               addrStoreIndex;

assign writeReady = (writeState == `STATE_READY);

assign port2Valid = (inputValidStore & writeReady) | (writeState == `STATE_FLUSHING);
assign port2WriteEnable = (inputValidStore & writeReady);

assign port2Address = (writeState != `STATE_FLUSHING) ? addressStore  :
                      {{(`ADDR_WIDTH-NUM_INDEX_BITS-2){1'b0}}, nextIndexToFlush, 2'b0};

assign addrStoreIndex = port2Address[NUM_INDEX_BITS+2-1:2];

assign port2ExistingAddressHomeCore = `HOME_CORE(port2ExistingAddr);
assign port2ExistingAddressHomeCoreMatches = port2ExistingAddressHomeCore[`CORE_ID_WIDTH-1:0] ==
                                             CORE_ID[`CORE_ID_WIDTH-1:0] ? 1'b1 : 1'b0;

assign port2AddressHomeCore = `HOME_CORE(port2Address);
```

```
assign port2AddressHomeCoreMatches = port2AddressHomeCore[`CORE_ID_WIDTH-1:0] ==
                                     CORE_ID[`CORE_ID_WIDTH-1:0] ? 1'b1 : 1'b0;

// If we are flushing, or just processing an eviction, initate the writeback to the cache hiearchy.
// Only if this core is the home core for an address.
assign initiateNewWriteback = (writeState == `STATE_FLUSHING && flushWalkDone == 1'b0 &&
        port2ExistingAddressHomeCoreMatches == 1'b1 && nextIndexBitToFlush != {NUM_SETS{1'b0}}) ? 1'b1 :
        writeReady == 1'b1 && port2Eviction == 1'b1 && port2ExistingAddressHomeCoreMatches == 1'b1 ? 1'b1 :
        1'b0;

/*****************
 Store port outputs
******************/

// Assume that stores are always consumed the cycle they arrive.
// From the core's point of view this is true, since they are sent
// in parallel with the memory write anyway.
assign requestCompleteStore = inputValidStore & writeReady;

assign finishedFlush = (writeState == `STATE_FLUSHING) && (flushWalkDone == 1'b1) &&
                       (evictionsAwaitingL1Confirmation == 1'b0) && (pendingEvictValid == 1'b0);

// Write back to L1 if the tag lookup indicated that there needs to be an eviction,
// and the evicted address's home core is this core. Initiate the writeback the same cycle as the tag lookup,
// though it might not be accepted for writeback until a subsequent cycle.
assign writebackValid = (pendingEvictValid == 1'b1) ? 1'b1 :
                        initiateNewWriteback == 1'b1 ? 1'b1 :
                        1'b0;

assign writebackAddr  = (pendingEvictValid == 1'b1) ? pendingEvictAddr :
                        initiateNewWriteback == 1'b1 ? port2ExistingAddr :
                        {`ADDR_WIDTH{1'b0}};

assign writebackData  = (pendingEvictValid == 1'b1) ? pendingEvictData :
                        initiateNewWriteback == 1'b1 ? port2ExistingData :
                        {`DATA_WIDTH{1'b0}};

/*************************
Store port state transitions
*************************/

integer c;
always @(posedge clk) begin
   // Reset all state between loop invocations
   if(reset == 1'b1) begin
      for(c=0; c < NUM_SETS; c = c + 1) begin
         ownerBitset[c] <= 1'b0;
```

```
    end
    writeState <= `STATE_READY;
    pendingEvictValid <= 1'b0;
    pendingEvictData <= {`DATA_WIDTH{1'b0}};
    pendingEvictAddr <= {`ADDR_WIDTH{1'b0}};
    evictionsAwaitingL1Confirmation <= {NUM_INDEX_BITS{1'b0}};
    flushWalkDone <= 1'b1;
end
else begin
    // If processing a new write...
    if(inputValidStore == 1'b1 && writeReady == 1'b1) begin
        ownerBitset[addrStoreIndex] <= port2AddressHomeCoreMatches;

        // Num outstanding evictions is this potential one,
        // minus any that might have come back this cycle
        evictionsAwaitingL1Confirmation <= initiateNewWriteback;

        // If something needed to be evicted, save the data and addr.
        if(initiateNewWriteback == 1'b1) begin
            pendingEvictData <= writebackData;
            pendingEvictAddr <= writebackAddr;
            pendingEvictValid <=  writebackValid;
            writeState <= `STATE_PENDING_EVICTION;
        end
        else begin
            writeState <= `STATE_READY;
        end
    end
    else if(writeState == `STATE_PENDING_EVICTION) begin
        // Num outstanding evictions is minus any that might have come back this cycle
        evictionsAwaitingL1Confirmation <= evictionsAwaitingL1Confirmation - writebackComplete;

        if(writebackAccepted == 1'b1) begin
            pendingEvictData <= {`DATA_WIDTH{1'b0}};
            pendingEvictAddr <= {`ADDR_WIDTH{1'b0}};
            pendingEvictValid <= 1'b0;
        end

        if(writeReady == 1'b1) begin
            writeState <= `STATE_READY;
        end
    end
    else if(writeState == `STATE_READY && startFlush == 1'b1) begin
        writeState <= `STATE_FLUSHING;
        flushWalkDone <= 1'b0;
    end
    else if(writeState == `STATE_FLUSHING) begin
```

```
        evictionsAwaitingL1Confirmation <= evictionsAwaitingL1Confirmation +
                            (initiateNewWriteback & (~pendingEvictValid)) - writebackComplete;

        // If there is nothing left to flush, set the walk finished flag.
        if(pendingEvictValid == 1'b0 && nextIndexBitToFlush == {NUM_SETS{1'b0}}) begin
           flushWalkDone <= 1'b1;
        end
        // If we haven't finished flushing, potentially writeback a new address,
        // as long as an existing one isn't pending acceptance.
        else if(pendingEvictValid == 1'b0 && flushWalkDone == 1'b0) begin
           // Invalidate ownership array
           ownerBitset[addrStoreIndex] <= 1'b0;

           // In the current implementation, the node always owns this address,
           // so this if will always be entered, unless the flush is ending.
           if(initiateNewWriteback == 1'b1) begin
              pendingEvictData <= writebackData;
              pendingEvictAddr <= writebackAddr ;
              pendingEvictValid <= writebackValid;
           end
        end
        else if (writebackAccepted == 1'b1) begin
           pendingEvictData <= {`DATA_WIDTH{1'b0}};
           pendingEvictAddr <= {`ADDR_WIDTH{1'b0}};
           pendingEvictValid <= 1'b0;
        end
        else if(finishedFlush == 1'b1) begin
           writeState <= `STATE_READY;
        end
     end
  end
end


/**************
 Load port logic
***************/

wire rcHit;
wire rcMiss;
wire issueL1Load;
wire l1LoadFinished;

assign readReady = (readState == `STATE_READY);

assign port1AddressHomeCore = `HOME_CORE(addressLoad);
```

```
assign port1AddressHomeCoreMatches = port1AddressHomeCore[`CORE_ID_WIDTH-1:0] ==
                                     CORE_ID[`CORE_ID_WIDTH-1:0] ? 1'b1 : 1'b0;

assign rcHit = (readState == `STATE_READY) && (inputValidLoad == 1'b1) && (port1Hit == 1'b1 ) ? 1'b1 : 1'b0;
assign rcMiss = (readState == `STATE_READY) && (inputValidLoad == 1'b1) && (port1Hit == 1'b0) ? 1'b1 : 1'b0;

// Only issue L1 load if this node is the address owner, or if it has never been stored before
assign issueL1Load = rcMiss & (port1AddressHomeCoreMatches | hashTableMiss);

assign l1LoadFinished = readState == `STATE_L1_LOAD && pendingL1LoadAccepted == 1'b1 &&
                        cacheLoadComplete == 1'b1;


/************************
 Load port output values
************************/

assign cacheLoadValid = (pendingL1LoadValid == 1'b1 && pendingL1LoadAccepted == 1'b0) ? 1'b1 :
                        (pendingL1LoadValid == 1'b1 && pendingL1LoadAccepted == 1'b1) ? 1'b0 :
                        (issueL1Load == 1'b1) ? 1'b1 :
                        1'b0;

assign cacheLoadAddr  = (pendingL1LoadValid == 1'b1 && pendingL1LoadAccepted == 1'b0) ? pendingL1LoadAddr :
                        (pendingL1LoadValid == 1'b1 && pendingL1LoadAccepted == 1'b1) ? {`ADDR_WIDTH{1'b0}} :
                        (issueL1Load == 1'b1) ? addressLoad :
                        {`ADDR_WIDTH{1'b0}};

assign dataOutLoad = rcHit ? port1DataOut :
                          l1LoadFinished ? cacheLoadResult :
                          {`DATA_WIDTH{1'b0}};

// The core considers a load as a hit if it was found locally, or loaded locally from its own L1.
// A miss implies the address needs to be fetched over the request network.
assign requestHitLoad = rcHit | l1LoadFinished;

// Inform the core that the load has completed, either found locally, loaded from L1 locally, or missed.
assign requestCompleteLoad = rcHit | l1LoadFinished |
                             (rcMiss & (~(port1AddressHomeCoreMatches | hashTableMiss)));


/**************************
 Load port state transitions
**************************/

always @(posedge clk) begin
   if(reset == 1'b1) begin
      readState <= `STATE_READY;
```

```
            pendingL1LoadValid <= 1'b0;
            pendingL1LoadAddr <= {`ADDR_WIDTH{1'b0}};
            pendingL1LoadAccepted <= 1'b0;
        end
        else begin
            if(readState == `STATE_READY) begin
                if(issueL1Load == 1'b1) begin
                    pendingL1LoadValid <= cacheLoadValid;
                    pendingL1LoadAddr <= cacheLoadAddr;
                    pendingL1LoadAccepted <= 1'b0;
                    readState <= `STATE_L1_LOAD;
                end
            end
            else if(readState == `STATE_L1_LOAD) begin
                if(cacheLoadAccepted == 1'b1) begin
                    pendingL1LoadAccepted <= 1'b1;
                end
                else if(l1LoadFinished) begin
                    pendingL1LoadValid <= 1'b0;
                    pendingL1LoadAddr <= {`ADDR_WIDTH{1'b0}};
                    pendingL1LoadAccepted <= 1'b0;
                    readState <= `STATE_READY;
                end
            end
        end
    end

endmodule
```

## B.8 priority_encoder.v

```
// Priority encoder, takes the rightmost bit of input and outputs
// the binary representation
// e.g.
//      4'b0101 -> 2'b00
//      4'b1110 -> 2'b01
//      4'b0100 -> 2'b10
//      4'b1000 -> 2'b11

// Used to reduce the number of memory words that need to be loaded
// to flush to the normal cache hierarchy from the ring cache.

module priority_encoder #(parameter NUM_SETS=2, parameter NUM_INDEX_BITS=1) (
      input wire [NUM_SETS-1:0]         ownerBitset,
      output wire [NUM_INDEX_BITS-1:0] nextIndexToFlush
);

   assign nextIndexToFlush = (ownerBitset[0]) ? 8'd0:
                             (ownerBitset[1]) ? 8'd1:
                             .
                             .
                             .
                             (ownerBitset[253]) ? 8'd253:
                             (ownerBitset[254]) ? 8'd254: 8'd255;
endmodule
```

## B.9 ARRAY.V

```
`include "defines.v"

// This module provides an interface to a memory array.

// It currently only supports a direct-mapped configuration, and models the memory
// array with an array of registers, which supports two combinational reads and two
// edge triggered writes per cycle. We currently use port1 for ring cache loads,
// so the write on port1 is unused. On port2, we use the combinational read to
// do a tag lookup, and the write to do the store itself.

// An SRAM implementation of this would require being double clocked to get the
// read timing we desire. Given the small size of the array, just using
// registers might be acceptable.

// We add two special outputs, existingData and existingAddr. These are the
// addresses/data that were already present in the area at a particular
// index. If the access was a hit, the existingAddress will match portXAddress.
// If the access was a miss, it will be the value of the address/data
// previous stored there. This is used for writing back evictions.

module array(
                    input wire                      reset,
                    input wire                      clk,

                    // Port 1 inputs
                    input wire                      port1Valid,
                    input wire [`ADDR_WIDTH-1:0]    port1Address,
                    input wire [`DATA_WIDTH-1:0]    port1WriteData,
                    input wire                      port1WriteEnable,

                    // Port 1 outputs
                    output wire [`DATA_WIDTH-1:0] port1DataOut,
                    output wire                     port1Hit,
                    output wire                     port1Complete,
                    output wire                     port1Eviction,
                    output wire [`DATA_WIDTH-1:0] port1ExistingData,
                    output wire [`ADDR_WIDTH-1:0] port1ExistingAddr,

                    // Port 2 inputs
                    input wire                      port2Valid,
                    input wire [`ADDR_WIDTH-1:0]    port2Address,
                    input wire [`DATA_WIDTH-1:0]    port2WriteData,
                    input wire                      port2WriteEnable,

                    // Port 2 outputs
```

```
                output wire [`DATA_WIDTH-1:0] port2DataOut,
                output wire                   port2Hit,
                output wire                   port2Complete,
                output wire                   port2Eviction,
                output wire [`DATA_WIDTH-1:0] port2ExistingData,
                output wire [`ADDR_WIDTH-1:0] port2ExistingAddr
);

    parameter NUM_ENTRIES = 256;
    parameter ASSOC = 1; // must be 1 for now

    localparam NUM_SETS = (NUM_ENTRIES / ASSOC);
    localparam NUM_INDEX_BITS = `CLOG2(NUM_SETS);
    localparam NUM_TAG_BITS   = `ADDR_WIDTH - 2 - NUM_INDEX_BITS;
    localparam WAY_ENTRY_SIZE =(ASSOC*(1+NUM_TAG_BITS+`DATA_WIDTH));

    // Data Array
    reg [WAY_ENTRY_SIZE-1:0]                array [0:NUM_SETS-1];

    // Input addr split into tag and index for both ports
    wire [NUM_TAG_BITS-1:0]                 port1AddrTag;
    wire [NUM_INDEX_BITS-1:0]               port1AddrIndex;
    wire [NUM_TAG_BITS-1:0]                 port2AddrTag;
    wire [NUM_INDEX_BITS-1:0]               port2AddrIndex;


    // Result of read split into entry, line, valid bit, tag, and data for both ports
    wire [1+NUM_TAG_BITS+`DATA_WIDTH-1:0]   port1ReadEntry;
    wire [WAY_ENTRY_SIZE-1:0]               port1ReadLine;
    wire                                    port1ReadValid;
    wire [NUM_TAG_BITS-1:0]                 port1ReadTag;
    wire [`DATA_WIDTH-1:0]                  port1ReadData;

    wire [1+NUM_TAG_BITS+`DATA_WIDTH-1:0]   port2ReadEntry;
    wire [WAY_ENTRY_SIZE-1:0]               port2ReadLine;
    wire                                    port2ReadValid;
    wire [NUM_TAG_BITS-1:0]                 port2ReadTag;
    wire [`DATA_WIDTH-1:0]                  port2ReadData;


    // The address that was actually stored in memory.  Same as the input address
    // if the memory lookup was a hit.
    wire [`ADDR_WIDTH-1:0]                  port1ReadAddr;
    wire [`ADDR_WIDTH-1:0]                  port2ReadAddr;

    // Does the loaded valud tags match the input address
    wire port1ReadTagsMatch;
```

```
wire port2ReadTagsMatch;

// Potential write split into entry and line.  Currently,
// port1 never does writes, but keep this anyway.
wire [1+NUM_TAG_BITS+`DATA_WIDTH-1:0]  port1WriteEntry;
wire [WAY_ENTRY_SIZE-1:0]              port1WriteLine;

// Potential write split into entry and line
wire [1+NUM_TAG_BITS+`DATA_WIDTH-1:0]  port2WriteEntry;
wire [WAY_ENTRY_SIZE-1:0]              port2WriteLine;

// Properties of input address
assign port1AddrIndex = port1Address[`ADDR_WIDTH-NUM_TAG_BITS-1:2];
assign port1AddrTag = port1Address[`ADDR_WIDTH-1:`ADDR_WIDTH-NUM_TAG_BITS];

// Result of read from memory array
assign port1ReadLine = array[port1AddrIndex];
assign port1ReadEntry = port1ReadLine[1+NUM_TAG_BITS+`DATA_WIDTH-1:0];
assign port1ReadValid = port1ReadEntry[WAY_ENTRY_SIZE-1];
assign port1ReadTag = port1ReadEntry[WAY_ENTRY_SIZE-2:`DATA_WIDTH];
assign port1ReadData = port1ReadEntry[`DATA_WIDTH-1:0];

// The address that was actually present in the assigned set
assign port1ReadAddr = {port1ReadTag, port1AddrIndex, 2'b00};

assign port1ReadTagsMatch = (port1ReadTag == port1AddrTag) ? 1'b1 : 1'b0;

// Potential write line to write to array
assign port1WriteEntry = {1'b1, port1AddrTag, port1WriteData};
assign port1WriteLine = port1WriteEntry;

// Properties of input address
assign port2AddrIndex = port2Address[`ADDR_WIDTH-NUM_TAG_BITS-1:2];
assign port2AddrTag = port2Address[`ADDR_WIDTH-1:`ADDR_WIDTH-NUM_TAG_BITS];

// Result of read from memory array
assign port2ReadLine = array[port2AddrIndex];
assign port2ReadEntry = port2ReadLine[1+NUM_TAG_BITS+`DATA_WIDTH-1:0];
assign port2ReadValid = port2ReadEntry[WAY_ENTRY_SIZE-1];
assign port2ReadTag = port2ReadEntry[WAY_ENTRY_SIZE-2:`DATA_WIDTH];
assign port2ReadData = port2ReadEntry[`DATA_WIDTH-1:0];

// The address that was actually present in the assigned set for this store
assign port2ReadAddr = {port2ReadTag, port2AddrIndex, 2'b00};

assign port2ReadTagsMatch = (port2ReadTag == port2AddrTag) ? 1'b1 : 1'b0;
```

```
    // Potential write line to write to array
    assign port2WriteEntry = {1'b1, port2AddrTag, port2WriteData};
    assign port2WriteLine = port2WriteEntry;


    // Outputs
    assign port1Hit = port1Valid & port1ReadValid & port1ReadTagsMatch;
    assign port1DataOut = port1Hit ? port1ReadData : {`DATA_WIDTH{1'b0}};
    assign port1Complete = port1Valid;
    assign port1Eviction = port1Valid & port1ReadValid & port1WriteEnable & (~port1Hit);
    assign port1ExistingData = port1Valid & port1ReadValid ? port1ReadData : {`DATA_WIDTH{1'b0}};
    assign port1ExistingAddr = port1Valid & port1ReadValid ? port1ReadAddr : {`ADDR_WIDTH{1'b0}};

    assign port2Hit = port2Valid & port2ReadValid & port2ReadTagsMatch;
    assign port2DataOut = port2Hit ? port2ReadData : {`DATA_WIDTH{1'b0}};
    assign port2Complete = port2Valid;
    assign port2Eviction = port2Valid & port2ReadValid & port2WriteEnable & (~port2Hit);
    assign port2ExistingData = port2Valid & port2ReadValid ? port2ReadData : {`DATA_WIDTH{1'b0}};
    assign port2ExistingAddr = port2Valid & port2ReadValid ? port2ReadAddr : {`ADDR_WIDTH{1'b0}};

    integer                             c;
    always@(posedge clk) begin
       if(reset == 1'b1) begin
          for(c=0; c < NUM_SETS; c = c + 1) begin
             array[c] <= {WAY_ENTRY_SIZE{1'b0}};
          end
       end
       else begin
          if(port1Valid & port1WriteEnable) begin
             array[port1AddrIndex] <= port1WriteEntry;
          end
          if(port2Valid & port2WriteEnable) begin
             array[port2AddrIndex] <= port2WriteEntry;
          end
       end
    end



endmodule
```

## B.10 bloom_filter.v

```
`include "hash.v"

// This module provides a/multiple hash function(s) to check if
// a memory address has already been stored to the ring cache.
// It supports two simultaneous address inputs - one for
// an address being loaded from the ring cache, for which
// we want to know whether it has been seen already or not, and one
// for an address being written to the ring cache, for which we want
// to record this fact in the hashtable.

// Seeds for the hash functions are random 32-bit numbers.

module bloom_filter(
                input wire                  reset,
                input wire                  clk,

                input wire [`ADDR_WIDTH-1:0] addrToCheck, // Return result based on this address
                input wire [`ADDR_WIDTH-1:0] addrToSet,   // Set the corresponding bits in the hashtable for
                input wire                  addrToSetValid, // this address.

                // The only output is whether the addrToCheck was absent
                // from the hashtable, which is all we care about.
                output wire                 hashTableMiss
);

   parameter BITS_IN_TABLE = 512;
   localparam INDEX_BITS = `CLOG2(BITS_IN_TABLE);

   // Bit array to record result of one or more hash functions.
   // Each hash function will set a bit in this table.  When checking for
   // membership, the coresponding bits for each hash function are checked,
   // and if both set, then the item has already been added.
   reg [BITS_IN_TABLE-1:0]              tableBitSet;

   wire [INDEX_BITS-1:0]                checkIndex1;
   wire [INDEX_BITS-1:0]                checkIndex2;

   wire [INDEX_BITS-1:0]                setIndex1;
   wire [INDEX_BITS-1:0]                setIndex2;

   wire                                 hash1IsSet;
   wire                                 hash2IsSet;


   hash #(.INDEX_BITS(INDEX_BITS)) hashCheck1 (
```

```
                                        .addr(addrToCheck),
                                        .hashSeed(32'h2abf7209),
                                        .out(checkIndex1)
                                        );

    hash #(.INDEX_BITS(INDEX_BITS)) hashCheck2 (
                                        .addr(addrToCheck),
                                        .hashSeed(32'h1a8fcee7),
                                        .out(checkIndex2)
                                        );


    hash #(.INDEX_BITS(INDEX_BITS)) hashSet1 (
                                        .addr(addrToSet),
                                        .hashSeed(32'h2abf7209),
                                        .out(setIndex1)
                                        );

    hash #(.INDEX_BITS(INDEX_BITS)) hashSet2 (
                                        .addr(addrToSet),
                                        .hashSeed(32'h1a8fcee7),
                                        .out(setIndex2)
                                        );


    // Check if both hash bits were already set.
    assign hash1IsSet = tableBitSet[checkIndex1];
    assign hash2IsSet = tableBitSet[checkIndex2];
    assign hashTableMiss = ~(hash1IsSet == 1'b1 && hash2IsSet == 1'b1);


    always@(posedge clk) begin
        if(reset == 1'b1) begin
            tableBitSet <= {BITS_IN_TABLE{1'b0}};
        end
        else begin
            if(addrToSetValid == 1'b1) begin
                tableBitSet[setIndex1] <= 1'b1;
                tableBitSet[setIndex2] <= 1'b1;
            end
        end
    end

endmodule
```

## B.11 HASH.V

```
// This module provides a multiply and shift hash function for 32-bit addresses.

// INDEX_BITS is the number of bits of output, i.e. the address size of the
// bitset where the result will be recorded/checked.

module hash #(parameter INDEX_BITS = 9) (
             input [`ADDR_WIDTH-1:0] addr, // Address to hash/
             input [`ADDR_WIDTH-1:0] hashSeed, // Random 32-bit hash seed.
             output [INDEX_BITS-1:0] out
);

   wire [(`ADDR_WIDTH*2)-1:0]        multResultHash;
   wire [`ADDR_WIDTH-1:0]            truncatedResult;
   wire [INDEX_BITS-1:0]             shiftedResult;


   assign multResultHash = addr * hashSeed;
   assign truncatedResult = multResultHash[`ADDR_WIDTH-1:0];
   assign shiftedResult = truncatedResult[`ADDR_WIDTH-1:`ADDR_WIDTH-INDEX_BITS];

   assign out = shiftedResult;

endmodule
```

## B.12 SIGNAL_BUFFER.V

```
`include "defines.v"
`include "signal_buffer_signal_tracker.v"

// This module implements all of the logic for the signal buffer.

// The compiler must know how many signal ids are available in the hardware
// before compilation.  The more signal ids available, the better potential performance.
// Each signal id has a coresponding bitset to track which cores have sent that signal,
// (and potentially how many times within a sliding window of iterations have they sent that signal).

// On every cycle, it can record up to `SIGNAL_BANDIWDTH number of signals, to different signal ids.
// It can also check if all the signals coresponding with a wait instruction have
// been received, so that the injecting core can enter a sequential segment.

// Signals from other cores as well as from the core attached to this RC node pass through the
// signal buffer.

// If the final signal to unblock a wait is received on the same cycle
// a core queries whether said wait can be unblocked, the release signal
// is raised that very cycle.

// The signal buffer has two special signal bitsets coresponding to two special wait/signal
// pairs that control the RC memory flush at the end of a loop invocation.  After the first
// pair is unblocked (signal id `NUM_SIGNALS - 1), the core may begin to flush.  After the memory
// finishes flushing, the next signal is sent (signal id `NUM_SIGNALS - 2) to inform all
// other cores of that fact.  When the wait instruction coresponding to this final signal
// is released, all cores know that all other cores have finished their flush.  The compiler
// must insert these special signals at the end of every loop invocation.

module signal_buffer(
                        input wire                                              clk,
                        input wire                                              reset,

                        // Used only to match up with C++ simulations for testing,
                        // since simulated phases might start on arbitrary iterations.
                        // It controls which signals each core records as having already been received.
                        input wire [`CORE_ID_WIDTH-1:0]                         firstIterationCoreId,

                        // Up to `SIGNAL_BANDWIDTH number of incoming signals recorded each cycle.
                        input wire [(`SIGNAL_BANDWIDTH * `SIGNAL_ENTRY_WIDTH)-1:0] incomingSignals,

                        // A wait ID the core is currently executing, and is checking whether
                        // the signals coresponding to this ID have been received from each core.
                        // Also whether the wait is 'light' and potentially skipable if it
                        // releasing it doesn't over/under flow the bit sets.
```

```
                    input wire                                          incomingWaitValid,
                    input wire [`ID_WIDTH-1:0]                          incomingWaitId,
                    input wire                                          incomingWaitLight,

                    // Output to tell the core the wait it is executing can be released,
                    // since the proper signals have already been received.
                    // A special wait release output is to tell the core it must flush the RC memory before
                    // releasing the wait instruction.
                    output wire                                         waitReleasedToStartFlush,
                    output wire                                         waitReleased
                    );

// See technical report for description.
// Corresponds to the size of the sliding window of iterations signals can
// potentially be received from.
parameter EPOCH_BOUND = 2;

parameter RECEIVER_CORE = 0;    // The core id of this RC node's core.

// only go up to NUM_SIGNALS-2, since special flush signals are the top two.
wire [`NUM_SIGNALS-2-1:0]                               waitReleasedFromCore;

// special wait release signal when the wait coresponding to the flush is released.
wire                                                    waitReleasedFromStartFlush;

// special wait release signal for when the final wait of the loop is released.
wire                                                    waitReleasedFromFinishLoop;

// Only one wait can be checked per cycle,
// so just OR the result of all signal releases together to find out if the core should proceed.
assign waitReleased = (|waitReleasedFromCore) | waitReleasedFromFinishLoop;

// Need a special output for the flush wait,
// so ring cache can hang on to it until the flush has started then finished
assign waitReleasedToStartFlush = waitReleasedFromStartFlush;


// Create one submodule for each possible signal.
// The compiler can limit the produced code to only having a specific
// number of signals.  Each submodule has N counters, where N=number of simulated cores.
// Note the -2 in the loop end condition,
// the highest two signal ids are reserved for the flush signals.
// Also note the incoming wait instruction to check is only valid if the ID matches the submodule.
genvar                                                  i;
generate
   for (i=0; i < `NUM_SIGNALS-2; i=i+1) begin : signalBitSets
      signal_tracker #(.EPOCH_BOUND(EPOCH_BOUND), .RECEIVER_CORE(RECEIVER_CORE), .SEGMENT_ID(i))
```

```
        signalBits (
                    .clk(clk),
                    .reset(reset),
                    .firstIterationCoreId(firstIterationCoreId),
                    .incomingSignals(incomingSignals),
                    .incomingWaitValid(incomingWaitValid && (incomingWaitId == i['ID_WIDTH-1:0])),
                    .incomingWaitLight(incomingWaitLight),
                    .waitReleased(waitReleasedFromCore[i])
                    );
       end
    endgenerate

    // Special flush/finish bits only need epoch bound 1,
    // since they by design don't allow cores to enter different epochs
    signal_tracker #(.EPOCH_BOUND(1), .RECEIVER_CORE(RECEIVER_CORE), .SEGMENT_ID(('NUM_SIGNALS-2)))
    signalFinishLoopBits (
                .clk(clk),
                .reset(reset),
                .firstIterationCoreId(firstIterationCoreId),
                .incomingSignals(incomingSignals),
                .incomingWaitValid(incomingWaitValid && (incomingWaitId == ('NUM_SIGNALS-2))),
                .incomingWaitLight(1'b0),
                .waitReleased(waitReleasedFromFinishLoop)
                );

    signal_tracker #(.EPOCH_BOUND(1), .RECEIVER_CORE(RECEIVER_CORE), .SEGMENT_ID(('NUM_SIGNALS-1)))
    signalStartFlushBits (
                .clk(clk),
                .reset(reset),
                .firstIterationCoreId(firstIterationCoreId),
                .incomingSignals(incomingSignals),
                .incomingWaitValid(incomingWaitValid  && incomingWaitId == ('NUM_SIGNALS-1)),
                .incomingWaitLight(1'b0),
                .waitReleased(waitReleasedFromStartFlush)
                );


endmodule
```

## B.13 signal_buffer_signal_tracker.v

```
`include "defines.v"
`include "signal_buffer_core_tracker.v"

// This module represents all the bits needed to track a single signal.  There are #ofCores submodules, since
// we need to track received signals for this id from all cores.

module signal_tracker(
                    input wire                                          clk,
                    input wire                                          reset,

                    // Used only to match up with C++ simulations for testing,
                     // since simualted phases might start on arbitrary iterations
                    input wire [`CORE_ID_WIDTH-1:0]                     firstIterationCoreId,

                    // Up to `SIGNAL_BANDWIDTH incoming signals to record this cycle.
                    input wire [(`SIGNAL_BANDWIDTH*`SIGNAL_ENTRY_WIDTH)-1:0] incomingSignals,

                    // If this input is high, see if enough signals have been received to let this core
                    // enter the sequential segment coresponding with this SEGMENT_ID of this submodule.
                    input wire                                          incomingWaitValid,
                    input wire                                          incomingWaitLight,

                    // If incomingWaitValid is high, raise this output high only if it is safe
                    // to enter the sequential segment.
                    output wire                                         waitReleased
                    );

   parameter EPOCH_BOUND = 2;     // Controls the size of the sliding window for signal tracking.
   parameter RECEIVER_CORE = 0;     // The core this signal buffer is attached to.
   parameter SEGMENT_ID = 0;    // The signal being tracked by these bitsets.


   // Breakout the different signal entry fields for the `SIGNAL_BANDWIDTH incoming signals,
   // to better check which are appropriate for this module.
   wire [`SIGNAL_ENTRY_WIDTH-1:0]                   breakoutSignals       [0:`SIGNAL_BANDWIDTH-1];
   wire                                             signalsValidBit       [0:`SIGNAL_BANDWIDTH-1];
   wire [`ID_WIDTH-1:0]                             signalsIds             [0:`SIGNAL_BANDWIDTH-1];
   wire [(`CORE_ID_WIDTH*`SIGNAL_BANDWIDTH)-1:0]    signalsOriginCoreIds;
   wire [`SIGNAL_BANDWIDTH-1:0]                     signalsMatchThisModule;

   // Check the bitsets for every core to see if a possible wait being
   // executed by the receiver core can be released.
   // Can only release if all cores have sent the proper signals,
   // so AND the wait release value from each core.
   wire [`NUM_CORES-1:0]                            waitReleasedFromCore;
```

```
assign waitReleased = &waitReleasedFromCore;

// Generate valid bits for the incoming signals,
// to mark which are appropriate to record for this signal ID,
// and having been sent from which core.
genvar i;
generate
   for (i=0; i < `SIGNAL_BANDWIDTH; i=i+1) begin : validSignals
      assign breakoutSignals[i] = incomingSignals[(`SIGNAL_ENTRY_WIDTH * (i+1))-1:
                                                  (`SIGNAL_ENTRY_WIDTH * (i))];

      assign signalsValidBit[i]    = breakoutSignals[i][`SIGNAL_ENTRY_WIDTH-1];

      assign signalsOriginCoreIds[(`CORE_ID_WIDTH*(i+1))-1:(`CORE_ID_WIDTH*i)] =
                signalsValidBit[i] == 1'b1 ?
             breakoutSignals[i][`SIGNAL_ENTRY_WIDTH-2:`ID_WIDTH] : {`CORE_ID_WIDTH{1'b0}};

      assign signalsIds[i] = signalsValidBit[i] == 1'b1 ?
                             breakoutSignals[i][`ID_WIDTH-1:0] : {`ID_WIDTH{1'b0}};

      // Only pass signals to submodule if the signal ids match the signal ID of this module
      assign signalsMatchThisModule[i] = signalsValidBit[i] == 1'b1 &&
                                         signalsIds[i] == SEGMENT_ID[`ID_WIDTH-1:0];
   end
endgenerate

// Generate signal tracking bits for each possible sending core.
// This generate also creates tracking bits for the receiver core, even though that is completely uncessary.
// Synthesis optimizes it away.
// As a parameter this submodule is assigned a sending core (SENDER_CORE).
// Only signals coresponding to SEGMENT_ID are set to valid.
// A special parameter coresponds to whether SEGMENT_ID is one of the special flush signals,
// which are initialized slightly differently.
generate
   for (i=0; i < `NUM_CORES; i=i+1) begin : coreBits
      core_tracker #(.EPOCH_BOUND(EPOCH_BOUND), .RECEIVER_CORE(RECEIVER_CORE),
                     .SENDER_CORE(i[`CORE_ID_WIDTH-1:0]),
                     .IS_FLUSH_SIGNAL((SEGMENT_ID == (`NUM_SIGNALS-1)) || (SEGMENT_ID == (`NUM_SIGNALS-2))))
      coreBits (
                .clk(clk),
                .reset(reset),
                .firstIterationCoreId(firstIterationCoreId),
                .incomingSignalsValid(signalsMatchThisModule),
                .incomingSignalsCoreIds(signalsOriginCoreIds),
                .incomingWaitValid(incomingWaitValid),
                .incomingWaitLight(incomingWaitLight),
                .waitReleased(waitReleasedFromCore[i])
```

```
                    );
        end
    endgenerate

endmodule
```

## B.14 signal_buffer_core_tracker.v

```
`include "defines.v"

// Corresponds to a single bitset, belonging to a single core, a single signal id,
// and reprsenting the signals received from a single core.

// Any value >= EPOCH_BOUND means a wait instruction for that signal ID can proceed.
// A counter fufilling this restriction implies that more signals have been received
// for this id from a particular sending core than have been sent by the receiver core,
// therefore the receiver core is safe to enter the sequential
// segment corresponding to the signal id.

// The compiler needs to guarentee, either through properties of the generated code,
// or by inserting light wait instructions, that the counter bits will never overflow or underflow.

// Whenever a core (that isn't the receiver) sends a signal coresponding to this module,
// the counter is incremented.
// When the receiver core of these bits sends a signal, the counter is decremented.

// Note that the tech report refers to states "-1", "0", "1", etc.
// The actual counting registers begin counting at 0,
// so state "-1" for an epoch_bound of 2 is recorded as 0 in the counter,
// state 0 recorded as 1, state 1 with 2, etc.

module core_tracker(
                                input wire                                      clk,
                                input wire                                      reset,

                                // The core that ran the first iteration of the loop. Is usually/always core 0.
                                input wire [`CORE_ID_WIDTH-1:0]                 firstIterationCoreId,

                                // The `SIGNAL_BANDWIDTH number of which cores sent these received signals, and
                                // whether they are valid.  This info is used to decided which signals
                                // should be recorded by this module.
                                input wire [(`CORE_ID_WIDTH*`SIGNAL_BANDWIDTH)-1:0] incomingSignalsCoreIds,
                                input wire [`SIGNAL_BANDWIDTH-1:0]              incomingSignalsValid,

                                // Whether we want to know if enough signals coresponding with the signal
                                // id and sending core have been received such that we can release
                                // the receiver core into the coresponding sequential segment.
                                input wire                                      incomingWaitValid,
                                input wire                                      incomingWaitLight,

                                // Combinatinal output as to whether a wait instruction is clear to proceed.
                                output reg                                      waitReleased
```

```
                    );


// For a value of 2, cores can drift 2 epochs of iterations apart.
parameter EPOCH_BOUND = 2;

// Which ring node this signal buffer belongs to.
parameter RECEIVER_CORE = 0;

// What sending core this set of signal tracking bits refers to.
// If == RECEIVER_CORE, this module is optimzed away.
parameter SENDER_CORE = 0;

// Special flush signals are initialized and treated slightly differently.
parameter IS_FLUSH_SIGNAL = 0;

// The intial value of the counters when the loop begins. Coresponds to state "0"
localparam INITIAL_VALUE = EPOCH_BOUND - 1;

// The value of the counters at which a normal wait instruction can proceed.  Coresponds to state "1"
localparam RELEASE_THRESH = EPOCH_BOUND;


// State bits

// EPOCH_BOUND*2 is the total number of required states, so 2 -> 2 bits
reg [(`CLOG2(EPOCH_BOUND*2)) - 1:0] counter;

// Combinational next counter value.
reg [(`CLOG2(EPOCH_BOUND*2)) - 1:0] nextCounter;


// Need to pre-set certain bits in signal buffer because on the first trip of iterations,
// signals are only expected from a subset of cores.
// Most of the complexity here is from starting a loop not on the first iteration (that is, on core 0).
// This is for phase simulation purposes.
// For a 'real' implementation, firstIterationCore would always equal core 0, so this logic simplifies.

always@(posedge clk) begin
   if(reset == 1'b1) begin
      // For flush signals, set all cores below the release threshold, since we want to wait for all cores.
      if(IS_FLUSH_SIGNAL == 1'b1) begin
         counter <= INITIAL_VALUE;
      end

      // The receiver core of these bits doesn't actually need to track his own signals,
      // but it's easier to assume that there are num bitsets == num cores,
```

```
      // so just preset the bits to always release waits.  Synthesis optimzies this away.
      else if(RECEIVER_CORE == SENDER_CORE) begin
         counter <= (EPOCH_BOUND*2)-1;
      end

      // Initialize all counters if not starting the first iteration.
      else if(RECEIVER_CORE != firstIterationCoreId) begin
         // If the core runs iterations in the same epoch as the first iteration core:
         if(RECEIVER_CORE > firstIterationCoreId) begin
            if(SENDER_CORE < firstIterationCoreId || SENDER_CORE > RECEIVER_CORE) begin
               counter <= RELEASE_THRESH;
            end
            else begin
               counter <= INITIAL_VALUE;
            end
         end
         // If this core is before the first iteration core
         // (that is, only starts running iterations from the second trip of iterations)
         else begin
            if(SENDER_CORE < firstIterationCoreId && SENDER_CORE > RECEIVER_CORE) begin
               counter <= RELEASE_THRESH;
            end
            else begin
               counter <= INITIAL_VALUE;
            end
         end
      end
      // If the owning core is the first core of the iteration,
      // it skips all of the first iteration wait instructions
      else begin
         counter <= RELEASE_THRESH;
      end
   end
   // If not reseting
   else begin
      counter <= nextCounter;
   end
end

// Record received signal logic
wire [`SIGNAL_BANDWIDTH-1:0] matchesReceiverCoreId;
wire [`SIGNAL_BANDWIDTH-1:0] matchesBitsCoreId;
wire                         anyMatchReceiverCoreId;
wire                         anyMatchBitsCoreId;

// For each possible incoming signal, check if it's valid
// (which means that it coresponds with the signal id coresponding to this module).
```

```
// This has been set by the parent module.
// Also check if the incoming signal either matches the sending core coresponding to this module,
// or if the receiver core of this whole signal buffer has sent the signal.
genvar i;
generate
   for (i=0; i < `SIGNAL_BANDWIDTH; i=i+1) begin : matchingSignals
      assign matchesReceiverCoreId[i] = incomingSignalsValid[i] == 1'b1 &&
                                        incomingSignalsCoreIds[(`CORE_ID_WIDTH*(i+1))-1:(`CORE_ID_WIDTH*i)]
                                        == RECEIVER_CORE ? 1'b1 : 1'b0;

      assign matchesBitsCoreId[i] = incomingSignalsValid[i] == 1'b1 &&
                                    incomingSignalsCoreIds[(`CORE_ID_WIDTH*(i+1))-1:(`CORE_ID_WIDTH*i)] ==
                                    SENDER_CORE ? 1'b1 : 1'b0;
   end
endgenerate

// For a particular set of bits corresponding to a signal id, and a sending core,
// we can only have two valid input signals.  This is a property of the HELIX execution model.

// One possible valid input corresponds to a signal sent by the sending core assigned to this bitset,
// and the other by the core that physically contains the bit sets.

// In the former case, we increment the bit counter to indicate that a signal was
// received from another core.  In the latter case, we decfement the counter to indicate that the
// physical receiver core of the bits has just left sent a signal.  Or if both cases are true, do both.

// If the received flush signal was sent by the same core, pretend it doesn't match
// and just increment the counter, for special flush signal semantics.
assign anyMatchReceiverCoreId = (|matchesReceiverCoreId) == 1'b1 && (IS_FLUSH_SIGNAL == 1'b0) ? 1'b1 : 1'b0;
assign anyMatchBitsCoreId = |matchesBitsCoreId;

always@(*) begin
   case ({anyMatchReceiverCoreId, anyMatchBitsCoreId})
     2'b00: nextCounter = counter;
     2'b01: nextCounter = counter+1;
     2'b10: nextCounter = counter-1;

     // For this case there are two possiblities.  One, SENDER_CORE==RECEIVER_CORE,
     // in which case we don't want to change the bits anyway.
     // When that comparison is false, then it means we've received two opposing signals,
     // so we also don't want to change the bits.

     2'b11: nextCounter = counter; // +1 and -1, so net effect of 0
   endcase
end
```

```
  // Wait release logic.  Release the wait based on the next counter value.
  always@(*) begin
    if(incomingWaitValid == 1'b1 && counter >= RELEASE_THRESH) begin
      waitReleased = 1'b1;
    end

    // If wait is 'light', can release as long as counter wouldn't potentially
    // underflow once the coresponding sequential segment executes, and the receiver core sends this signal.
    // If the counter was 0, and we released this wait, it will underflow once the
    // next receiver signal is sent.
    else if(incomingWaitValid == 1'b1 && counter > {(`CLOG2(EPOCH_BOUND*2)){1'b0}} &&
                                                    incomingWaitLight == 1'b1) begin

      waitReleased = 1'b1;
    end
    else begin
      waitReleased = 1'b0;
    end
  end

endmodule
```

# References


[1] (2013). *Tile Processor Architecture Overview for the TILEPro Series*. Tilera Corportation.

[2] Allen, R. & Kennedy, K. (2002). *Optimizing compilers for modern architectures*. Morgan Kaufmann.

[3] Barroso, L. A., Clidaras, J., & Hölzle, U. (2013). The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3), 1–154.

[4] Bernstein, A. (1966). Analysis of programs for parallel processing.

[5] Borkar, S. & Chien, A. A. (2011). The future of microprocessors. *Commun. ACM*, 54(5), 67–77.

[6] Breach, S. E., Vijaykumar, T. N., & Sohi, G. S. (1994). The anatomy of the register file in a multiscalar processor. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27 (pp. 181–190). New York, NY, USA: ACM.

[7] Burger, D., Goodman, J. R., & Kägi, A. (1996). Memory bandwidth limitations of future microprocessors. In *ISCA*.

[8] Campanoni, S., Agosta, G., Reghizzi, S. C., & Biagio, A. D. (2010). A Highly Flexible, Parallel Virtual Machine: Design and Experience of ILDJIT. In *Software: Practice and Experience*.

[9] Campanoni, S., Brownell, K., Kanev, S., Jones, T., Wei, G.-Y., & Brooks, D. (2014). Helix-rc: An architecture-compiler co-design for automatic parallelization of irregular programs. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on* (pp. 217–228).

[10] Campanoni, S., Holloway, G., Wei, G.-Y., & Brooks, D. (2015). Helix-up: Relaxing program semantics to unleash parallelization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15 (pp. 235–245). Washington, DC, USA: IEEE Computer Society.

[11] Campanoni, S., Jones, T., Holloway, G., Reddi, V. J., Wei, G.-Y., & Brooks, D. (2012a). Helix: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12 (pp. 84–93). New York, NY, USA: ACM.

[12] Campanoni, S., Jones, T. M., Holloway, G., Janapa Reddi, V., Wei, G.-Y., & Brooks, D. (2012b). HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing. In *CGO*.

[13] Campanoni, S., Jones, T. M., Holloway, G., Wei, G.-Y., & Brooks, D. (2012c). HELIX: Making the Extraction of Thread-Level Parallelism Mainstream. In *IEEE Micro*.

[14] Chatterjee, R., Ryder, B. G., & Landi, W. A. (1999). Relevant Context Inference. In *POPL*.

[15] Chrysos, G. (2012). Knights corner, intel's first many integrated core (mic) architecture product. In *Hot Chips*, volume 24.

[16] Cytron, R. (1986). DOACROSS: Beyond vectorization for multiprocessors. In *ICPP*.

[17] Danowitz, A., Kelley, K., Mao, J., Stevenson, J. P., & Horowitz, M. (2012). Cpu db: Recording microprocessor history. *Queue*, 10(4), 10:10–10:27.

[18] Dennard, R. H., Rideout, V., Bassous, E., & Leblanc, A. (1974). Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5), 256–268.

[19] Deutsch, A. (1992). A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *ICCL*.

[20] Gao, C., Gutierrez, A., Dreslinski, R. G., Mudge, T., Flautner, K., & Blake, G. (2014). A study of thread level parallelism on mobile devices. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on* (pp. 126–127).: IEEE.

[21] Gratz, P., Kim, C., Sankaralingam, K., Hanson, H., Shivakumar, P., Keckler, S. W., & Burger, D. (2007a). On-Chip Interconnection Networks of the TRIPS Chip. In *IEEE Micro*.

[22] Gratz, P., Sankaralingam, K., Hanson, H., Shivakumar, P., McDonald, R., Keckler, S., & Burger, D. (2007b). Implementation and evaluation of a dynamically routed processor operand network. In *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on* (pp. 7–17).

[23] Guo, B., Bridges, M. J., Triantafyllis, S., Ottoni, G., Raman, E., & August, D. I. (2005). Practical and accurate low-level pointer analysis. In *CGO*.

[24] Hamerly, G., Perelman, E., & Calder, B. (2004). How to use simpoint to pick simulation points. In *ACM SIGMETRICS Performance Evaluation Review*.

[25] Hammond, L., Hubbert, B. A., Siu, M., Prabhu, M. K., Chen, M. K., & Olukotun, K. (2000). The Stanford Hydra CMP. In *IEEE Micro*.

[26] Hammond, L., Willey, M., & Olukotun, K. (1998). Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII (pp. 58–69). New York, NY, USA: ACM.

[27] Huang, J., Raman, A., Jablin, T. B., Zhang, Y., Hung, T.-H., & August, D. I. (2010). Decoupled software pipelining creates parallelization opportunities. In *CGO*.

[28] Hurson, A. R., Lim, J. T., Kavi, K. M., & Lee, B. (1997). Parallelization of doall and doacross loops - a survey. In *Advances in Computers*, volume 45.

[29] Jaleel, A. (2007). Memory characterization of workloads using instrumentation-driven simulation – a pin-based memory characterization of the spec cpu2000 and spec cpu2006 benchmark suites. In *Technical Report, VSSAD, Intel Corporation*.

[30] Jerger, N. E. & Peh, L.-S. (2009). *On-Chip Networks*. Synthesis Lectures on Computer Architecture. Morgan & Claypool.

[31] Johnson, T. A., Eigenmann, R., & Vijaykumar, T. N. (2007). Speculative thread decomposition through empirical optimization. In *PPoPP*.

[32] Kanev, S., Wei, G.-Y., & Brooks, D. (2012a). XIOSim: power-performance modeling of mobile x86 cores. In *ISLPED*.

[33] Kanev, S., Wei, G.-Y., & Brooks, D. (2012b). Xiosim: Power-performance modeling of mobile x86 cores. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12 (pp. 267–272). New York, NY, USA: ACM.

[34] Kim, C., Sethumadhavan, S., Govindan, M. S., Ranganathan, N., Gulati, D., Burger, D., & Keckler, S. W. (2007). Composable lightweight processors. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on* (pp. 381–394).: IEEE.

[35] Kim, H., Johnson, N. P., Lee, J. W., Mahlke, S. A., & August, D. I. (2012). Automatic speculative doall for clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12 (pp. 94–103). New York, NY, USA: ACM.

[36] Kim, N., Austin, T., Baauw, D., Mudge, T., Flautner, K., Hu, J., Irwin, M., Kandemir, M., & Narayanan, V. (2003). Leakage current: Moore's law meets static power. *Computer*, 36(12), 68–75.

[37] Kistler, M., Perrone, M., & Petrini, F. (2006). Cell multiprocessor communication network: Built for speed. 26(3).

[38] Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., & Torrellas, J. (2006). POSH: A TLS compiler that exploits program structure. In *PPoPP*.

[39] Mars, J., Tang, L., Hundt, R., Skadron, K., & Soffa, M. L. (2011). Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture* (pp. 248–259).: ACM.

[40] Martin, M. M. K. (2003). *Token coherence*. PhD thesis, University of Wisconsin-Madison.

[41] Mehrara, M., Hao, J., Hsu, P.-C., & Mahlke, S. (2009). Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09 (pp. 166–176). New York, NY, USA: ACM.

[42] Moore, G. (1965). Cramming more components onto integrated circuits. *Electronics Magazine*.

[43] Muralimanohar, N., Balasubramonian, R., & Jouppi, N. P. (2009). *CACTI 6.0: A tool to model large caches*. Technical Report 85, HP Laboratories.

[44] Nicolau, A., Li, G., & Kejariwal, A. (2009a). Techniques for efficient placement of synchronization primitives. In *PPoPP*.

[45] Nicolau, A., Li, G., Veidenbaum, A. V., & Kejariwal, A. (2009b). Synchronization optimizations for efficient execution on multi-cores. In *ICS*.

[46] Ottoni, G., Rangan, R., Stoler, A., & August, D. I. (2005). Automatic thread extraction with decoupled software pipelining. In *MICRO*.

[47] Prabhu, M. K. & Olukotun, K. (2005). Exposing speculative thread parallelism in spec2000. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05 (pp. 142–152). New York, NY, USA: ACM.

[48] Raman, A., Kim, H., Mason, T. R., Jablin, T. B., & August, D. I. (2010). Speculative parallelization using software multi-threaded transactions. In *ASPLOS*.

[49] Raman, E., Ottoni, G., Raman, A., Bridges, M. J., & August, D. I. (2008). Parallel-stage decoupled software pipelining. In *CGO*.

[50] Rangan, R. et al. (2004). Decoupled software pipelining with the synchronization array. In *PACT*.

[51] Robatmil, B., Li, D., Esmaeilzadeh, H., Govindan, S., Smith, A., Putnam, A., Burger, D., & Keckler, S. W. (2013). How to Implement Effective Prediction and Forwarding for Fusable Dynamic Multicore Architectures. In *HPCA*.

[52] Rosenfeld, P., Cooper-Balis, E., & Jacob, B. (2011). DRAMSim2: A Cycle Accurate Memory System Simulator. In *IEEE Computer Architecture Letters*.

[53] Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Ranganathan, N., Burger, D., Keckler, S. W., McDonald, R. G., & Moore, C. R. (2004). TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP. In *ACM TACO*.

[54] Sohi, G. S., Breach, S. E., & Vijaykumar, T. N. (1995). Multiscalar processors. In *ISCA*.

[55] Sorin, D. J., Hill, M. D., & Wood, D. A. (2011). A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3), 1–212.

[56] Steffan, J. G., Colohan, C., Zhai, A., & Mowry, T. C. (2005). The STAMPede approach to thread-level speculation. In *ACM Transactions on Computer Systems*.

[57] Steffan, J. G., Colohan, C. B., Zhai, A., & Mowry, T. C. (2002). Improving value communication for thread-level speculation. In *HPCA*.

[58] Taylor, M. B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffman, H., Johnson, P., Lee, J.-W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., & Ararwal, A. (2002). The RAW microprocessor: A computational fabric for software circuits and general-purpose programs. In *IEEE Micro*.

[59] Taylor, M. B., Lee, W., Amarasinghe, S. P., & Agarwal, A. (2005). Scalar Operand Networks. In *IEEE Transactions on Parallel Distributed Systems*.

[60] Tournavitis, G., Wang, Z., Franke, B., & O'Boyle, M. F. P. (2009). Towards a holistic approach to auto-parallelization. In *PLDI*.

[61] Vachharajani, N. et al. (2007). Speculative decoupled software pipelining. In *PACT*.

[62] Van der Wijngaart, R. F., Mattson, T. G., & Haas, W. (2011). Light-weight communications on intel's single-chip cloud computer processor. *ACM SIGOPS Operating Systems Review*, 45(1), 73–83.

[63] Vijaykumar, T. & Sohi, G. S. (1998). Task selection for a multiscalar processor. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture* (pp. 81–92).: IEEE Computer Society Press.

[64] Wentzlaff, D., Griffin, P., Hoffmann, H., Bao, L., Edwards, B., Ramey, C., Mattina, M., Miao, C.-C., Brown, III, J. F., & Agarwal, A. (2007). On-chip interconnection architecture of the tile processor. In *IEEE Micro*.

[65] Zhai, A., Colohan, C. B., Steffan, J. G., & Mowry, T. C. (2002). Compiler optimization of scalar value communication between speculative threads. In *ASPLOS*.

[66] Zhai, A., Steffan, J. G., Colohan, C. B., & Mowry, T. C. (2008). Compiler and hardware support for reducing the synchronization of speculative threads. In *ACM TACO*.

[67] Zhong, H., Mehrara, M., Lieberman, S., & Mahlke, S. (2008). Uncovering hidden loop level parallelism in sequential applications. In *HPCA*.