Technical University of Munich

School of Computation, Information and Technology

-- Informatics --

Master's Thesis in Information Systems

Generating Heterogeneous Cache Coherence Protocols for gem5 Simulations

Simon Dittrich

Technical University of Munich

School of Computation, Information and Technology
-- Informatics --

Master's Thesis in Information Systems

Generating Heterogeneous Cache Coherence Protocols for gem5 Simulations

Generierung heterogener Cache-Kohärenz-Protokolle für gem5-Simulationen

Author:	Simon Dittrich
Supervisor:	Prof. Dr. Pramod Bhatotia
Advisors:	Dr. Anatole Lefort
Start Date:	15.03.2024
Submission Date:	20.10.2024

I confirm that this Master's thesis is my own work and I have documented all sources and material used.

Munich, 20.10.2024 Simon Dittrich

Acknowledgements

I would like to thank my supervisor, Prof. Dr. Pramod Bhatotia, for offering me the opportunity to work at his chair. I also want to thank my advisor, Dr. Anatole Lefort, for his continuous assistance with my work and willingness to answer all my questions. Finally, I would like to thank my family and friends for their support and encouragement during this journey.

Abstract

Ensuring cache coherence in heterogeneous systems requires complex bridging logic between the individual protocols. Quantifying the performance impact of this bridging logic using analytical methods alone is difficult, as it is highly dependent on application-specific access patterns and the complex interactions between different protocols.

To address this, we propose the use of simulations, specifically via gem5, to analyze the performance impact of combining various cache coherence protocols and identify commonalities between them. Given the large number of potential protocol combinations, manually configuring simulation setups becomes impractical. To overcome this, we developed a generator capable of automatically creating heterogeneous simulation environments based on high-level input descriptions.

In this work, we present a flexible gem5 setup for heterogeneous systems alongside a code generator that automatically synthesizes the necessary protocol definitions to run simulations with a wide variety of protocol combinations.

Zusammenfassung

Die Gewährleistung von Cache-Kohärenz in heterogenen Systemen erfordert komplexe Logik, um Unterschiede zwischen den einzelnen Protokollen zu überbrücken. Es ist schwierig, die Auswirkung dieser Überbrückungslogik auf die Leistung allein mit analytischen Methoden zu quantifizieren, da sie in hohem Maße von anwendungsspezifischen Zugriffsmustern und den komplexen Wechselwirkungen zwischen verschiedenen Protokollen abhängt.

Daher schlagen wir den Einsatz von Simulationen vor, insbesondere über gem5, um die Leistungsauswirkungen der Kombination verschiedener Cache-Kohärenzprotokolle zu analysieren und Gemeinsamkeiten zwischen ihnen zu ermitteln. Angesichts der großen Anzahl möglicher Protokollkombinationen ist die manuelle Konfiguration von Simulations-Setups unpraktisch. Um dieses Problem zu lösen, haben wir einen Generator entwickelt, der in der Lage ist, automatisch heterogene Simulationsumgebungen auf der Grundlage von High-Level-Eingaben zu erstellen.

In dieser Arbeit stellen wir ein flexibles gem5-Setup für heterogene Systeme sowie einen Code-Generator vor, der automatisch die notwendigen Protokolldefinitionen synthetisiert, um Simulationen mit einer Vielzahl von Protokollkombinationen durchzuführen.

Contents

1 Introduction 10
2 Background 12
2.1 Generic Cache System Model 12
2.1.1 Cache Coherence Protocols 12
2.1.2 Inter-Controller Communication 13
2.2 Realistic Cache System Model 14
2.2.1 Additional Cache States 14
2.2.2 Hierarchical Systems 15
2.2.3 Heterogenous Systems 16
2.3 Generating Cache Coherence Protocols 17
2.3.1 ProtoGen 18
2.3.2 HieraGen 18
2.3.3 HeteroGen 19
2.4 Simulating Systems with gem5 20
2.5 Non Uniform Memory Access 22
3 Motivation 24
4 Design 25
4.1 Architecture of a Cluster with Disaggregated Memory 25
4.2 Heterogeneity in the System architecture 26
4.2.1 Heterogeneity via Fused Directory 26
4.2.2 Heterogeneity via Bridge 27
4.3 Automatic Synthesis of Required Protocols 28
4.3.1 Protocol Specification for Nodes 28
4.3.2 Protocol Generation from High Level Descriptions 29
4.3.3 Integration with gem5 29
4.4 Reusable Simulator Setup 30
5 Implementation 31

5.1 Configuration of gem5 31
5.1.1 Cores and Memory 31
5.1.2 Ruby Caches 32
5.1.3 Garnet Interconnect 34
5.1.4 Simulated Workload 35
5.1.5 Execution Results 36
5.2 Supporting NUMA Awareness 36
5.2.1 Physical Address Interleaving 36
5.2.2 Explicitly Mapped Memory 37
5.3 Custom Cache Coherence Protocols with Ruby 38
5.3.1 Defining Messages 39
5.3.2 Declaring Components 39
5.3.3 Sending And Receiving Messages 42
5.3.4 Directory Implementation 44
5.4 Simulating Heterogeneous Systems 45
5.4.1 Extending Machine IDs 45
5.4.2 Merging Messages 46
5.4.3 Communication Patterns for Bridges 46
5.4.4 Clustered System Configuration 47
5.5 Ruby Protocol Generation 49
5.5.1 Bridge Generator 49
5.5.1.1 Input Specification 49
5.5.1.2 Immediate Output 51
5.5.2 Basic Structures 52
5.5.3 Event Handlers 52
5.5.3.1 General Structure 53
5.5.3.2 Allocating Cache Entries 54
5.5.3.3 Dealing with Missing Virtual Channel Information 54
5.5.3.4 Dealing with Implicit Memory Operations 55
5.5.4 Generating Bridge Controllers 55
5.5.5 Message Flattening 56

5.5.6 Order-preserving Stalling 57
5.5.7 Functional Access Permissions 58
5.6 Additional Topics 59
5.6.1 Functional Access Functions 59
5.6.2 Stalling Functions 60
5.6.2.1 Vertical Stalling 60
5.6.2.2 Message Recycling 61
5.6.2.3 Blocking Addresses 61
5.6.3 Request Routing 61
6 Evaluation 63
6.1 Setup 63
6.2 Performance of Generated Flat Protocols 63
6.3 Performance Impact of Preserving Message Ordering 65
6.4 Performance of Heterogenous Protocols 66
6.4.1 Fused Directory 66
6.4.2 Bridge 67
6.5 Correctness 67
7 Summary 68
8 Future Work 69
8.1 Testing of the Bridge Controller 69
8.2 More Benchmarks 69
8.3 Improved Code Quality of the Generated SLICC Code 69
List of Figures 71
List of Listings 72
Appendix A: Splash-4 Parameterization 73
Bibliography 74

1 Introduction

In large-scale cloud data centers, memory constitutes a significant portion of operational costs [1]. As a result, the CXL [2] standard was developed to enable memory disaggregation and allow cloud providers to consolidate system memory into dedicated nodes. Consolidating all memory into a single location reduces resource wastage due to uneven resource utilization on individual servers and would additionally allow for shared memory communication between nodes. However, many cloud providers also offer an increasing number of compute architectures that differ from the traditional x86 CPUs, such as ARM CPUs, or dedicated accelerators, such as GPUs, TPUs, or FPGAs. Many of these alternate architectures come with memory access semantics that can differ drastically from those on x86-based platforms, which makes integrating them with the CXL-based memory pool problematic.

A key challenge is how to handle cache coherence in such a heterogeneous environment. Traditional coherence protocols are designed with homogenous systems in mind and, therefore, require additional bridging logic to integrate them into a network with other fundamentally different protocols. It is difficult to quantify the performance impact of this additional bridging logic using purely analytical methods as it likely heavily depends on application-specific access patterns and the complex interplay between the individual cache coherence protocols.

We believe that simulations are the key to analyzing and characterizing this performance impact. More specifically, we want to use gem5 to run various well-known benchmarks using multiple protocol combinations. However, the many possible protocol combinations make manually creating a simulation setup for each one an infeasible solution. Towards that end, we built a generator that can automatically generate heterogeneous simulation setups based on high-level inputs.

In summary, the contributions of this thesis are as follows.

- We present a gem5 configuration that replicates a cluster with disaggregated memory
- We present a generator capable of automatically generating the SLICC code required to model the combination of arbitrary cache coherence protocols from high-level descriptions
- We measured the performance of our generated protocols against the performance of the handwritten protocols from gem5′s standard library when running the Splash-4 benchmark suite

2 Background

2.1 Generic Cache System Model

2.1.1 Cache Coherence Protocols

As CPU clock speed increased, the access latencies of the main memory became a significant bottleneck. To hide these latencies, many architectures added caches between the processor and the main memory, which mirror part of the main memory at significantly reduced access cost. However, as core counts continued to increase, interconnect traffic became an issue, and CPU makers started to induce private caches in front of each core.

When multiple private caches can contain copies of the same region of memory, they must coordinate their access to avoid losing data once each cache writes its data back to the main memory. Formally, this requirement can be condensed into two invariants that must be upheld [3]:

1. **Single-Writer, Multiple-Read (SWMR):** For any memory location A, at any given time, there exists only a single core that may write to A (and can also read it) or some number of cores that may only read A.
2. **Data-Value:** The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch.

On a practical level, access permissions are not managed at one-byte granularity but rather for larger blocks known as cache lines. On modern systems, a cache line typically has a size of 64 bytes [4, Section 12.1]. The individual access permissions of each cache line are kept coherent by the cache coherence protocol. The basis for many widely used cache coherence protocols is the MSI protocol [5]. In the MSI protocol, each cache line in each cache can be in one of three states:

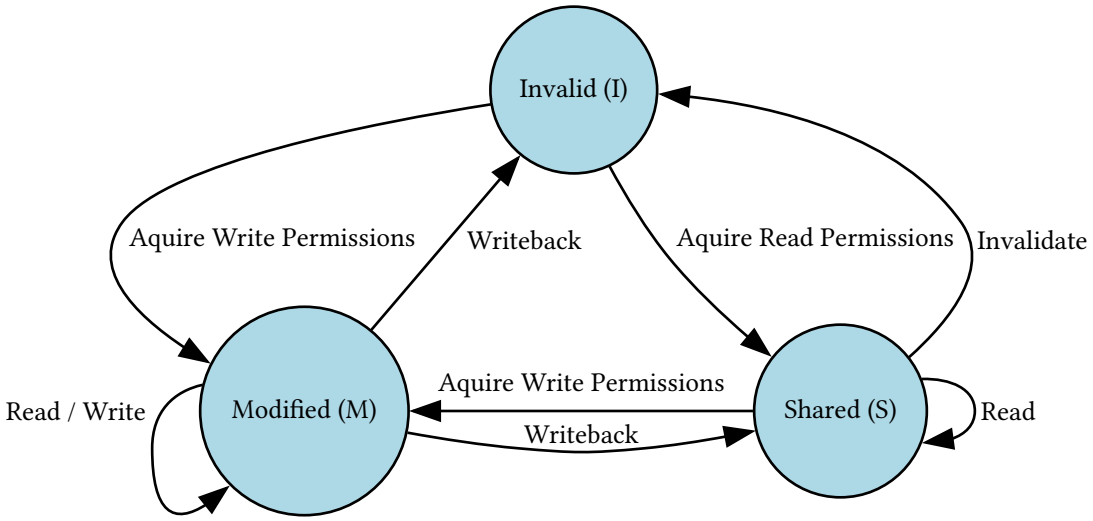


Figure 1: The basic state machine of a MSI cache

- **Modified (M):** The cache line is present in the current cache and can be read or written. The current cache is the only cache containing this specific cache line.
- **Shared (S):** The cache line is present in the current cache but can only be read. The current cache line can also be present in other caches.
- **Invalid (I):** The cache line is not present in the current cache.

All cache lines are initially in the I state. When attempting to read a cache line that is currently in the I state, the cache must first acquire read permissions for it. This process will force any cache line currently in the M state to downgrade to either S or I by propagating its changes back to memory. Conversely, attempting to write to a cache line currently in either S or I will cause it to transition into the M state and force all other caches into the I state. Downgrading from S to I typically does not require a write-back, as the S state does not allow for modifications.

2.1.2 Inter-Controller Communication

Acquiring access permission requires communication between different caches in the system. This communication is coordinated by a *Directory* [5]. While private caches are the interface of CPU cores from a cache system perspective, a directory is the interface of the memory controller. The directory tracks the states of all cache lines for all caches. For this reason, the directory maintains both the owner

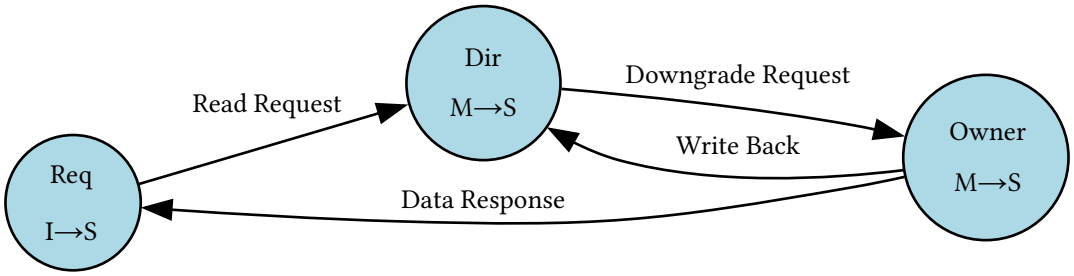


Figure 2: A coherence flow for acquiring read permissions.

and a list of shares for all cache lines. If the directory state is in M then the owner is the sole cache in M. If the directory is in S then the shares list contains all caches that share the S state.

When a cache requires more permissions for a specific cache line than it currently possesses, it initiates the upgrade process by sending a message to the directory responsible for the cache line. The directory then consults its internal data structures to identify which caches need to be downgraded to maintain the SWMR invariant. It then dispatches the relevant messages to these caches. After all relevant caches confirm their downgrade, the original requestor can transition into its desired state with stronger access rights.

Another important function of the directory is that it imposes a total order on all ongoing transactions for any given cache line, as all transactions start by consulting a directory. This is important to satisfy the Data-Value invariant.

It is worth mentioning that cache coherence protocols can be implemented without a directory using a technique known as snooping. However, doing so results in higher bandwidth demands for the interconnect. For this reason, this technique is rarely used for the complex multicore systems this work focuses on.

2.2 Realistic Cache System Model

2.2.1 Additional Cache States

Real-world systems rarely use the MSI protocol in its base form but typically augment it with additional states to better handle common access patterns.

Intel Processors [4, Section 12.4], for example, uses a MESI protocol for cache coherence. The additional E state (*Exclusive*) is entered when a cache attempts to acquire read permission for a cache line that has no other sharers. On the surface level, the E state is equivalent to the S state, except it can be silently upgraded into M without communicating with the directory. This state reduces the number of required coherence transactions in the common read-modify-write access pattern [5].

Most ARM CPU architectures [6] use the MOESI protocol, which extends the MESI protocol with an additional O state (*Owned*). The O state signifies that the current cache is the owner of the current cache line and holds the most up-to-date version of it. If a cache holds a cache line in the O state, then all other sharers are referencing this version of the data rather than the data in the main memory. This addition of this state allows the protocol to reduce the number of writes to the main memory [5].

2.2.2 Hierarchical Systems

Additional states are not the only optimizations modern systems introduced to improve the basic MSI model. Another difference is that realistic systems are often hierarchical and have more than one cache level [7]. The main reason for opting for a hierarchical protocol instead of a flat one is that flat protocols tend to have difficulty scaling to high core counts. In a flat protocol with a single directory, said directory is the sole coherence point, and all coherence requests must flow through it. This bottleneck and the associated demands on the interconnect become increasingly tricky problems to solve as core counts increase. Introducing shared caches into the system can help with this issue, as the shared caches become the coherence points for the private caches, effectively taking the role of the directory and splitting the load between them. The shared cache can then either fulfill the request directly or forward it to the directory.

Another advantage of shared caches with respect to cache coherence is that the protocols used at each level are largely independent of one another. This allows each level of the cache hierarchy to be optimized based on its own requirements,

such as trading increased latency for reduced bandwidth usage. For instance, a shared cache may prioritize reducing bandwidth usage to minimize interconnect traffic between clusters. In contrast, a private cache may focus more on minimizing latency to ensure fast access times to frequently used data. Additionally, this independence can help simplify the implementation and validation of cache coherence protocols, as each level can be optimized or replaced without requiring a complete overhaul of the entire memory hierarchy. This modularity allows for adaptable system architectures, as varying amounts of base components can be combined to meet the desired performance targets without requiring any modifications to the underlying architecture.

2.2.3 Heterogenous Systems

Another aspect of modern systems is that they are also increasingly heterogeneous. One example of this trend is the rising number of coprocessors present in many systems, such as Graphics Processing Units (GPUs), Digital Signal Processors (DSPs), Tensor Processing Units (TPUs), or even Field-Programmable Gate Arrays (FPGAs). While many of the tasks solved by these coprocessors could also be solved using the CPU, doing so generally leads to significantly reduced performance and power consumption compared to specialized hardware.

However, offloading work to a coprocessor requires the coprocessor to have access to the data it is supposed to work on. Traditionally, many coprocessors operate on separate memory from the main processor. Consequently, dispatching work to a coprocessor often involves first copying the required data between memory regions. While this strong separation between memory regions comes with advantages, such as being able to choose the optimal memory technology for each processor type (high bandwidth memory for GPUs and low latency memory for CPUs) and no need to care about coherence, it also heavily limits how finely a workload can be distributed between mixed workloads and leads to poor memory utilization for one-sided workloads.

For this reason, there is a growing demand for system architectures where memory can be seamlessly shared between different kinds of processors. To meet

this demand, several cache coherence interconnect standards, such as AMBA CHI [8] or CXL [2] were developed. However, providing cache coherence in such a system is a significant challenge, as coprocessors tend to have significantly weaker consistency models than CPUs because their limited area of use allows for more specialized designs. These weaker consistency models tend to be accompanied by bespoke cache coherence protocols that differ significantly from the generalist protocols used by CPUs. As a result, additional bridging logic is necessary to fuse the system into a unified heterogeneous whole [8], [9].

However, heterogeneity is not just limited to accelerators. The CXL standard, for example, defines MESI as its cache-coherent protocol. Attempting to communicate over it using an ARM processor that internally uses the MOESI protocol also results in a heterogeneous system that also requires bridging logic to function correctly.

2.3 Generating Cache Coherence Protocols

The cache coherence protocols presented so far only contained stable states, as their transactions were assumed to be atomic. This is rarely the case in practice, as an atomic bus is hard to scale to high core counts. For this reason, practical implementations generally have to augment the set of stable states with numerous transient states to reflect the state of a cache line during a partially executed transaction.

The Requestor, in Figure 2, for example, would generally not be able to directly transition from I to S but would instead transition into a transient state IS after sending its request and transition from IS to S upon receiving the response. However, the data response will not be guaranteed to be the only message to arrive during this transient state.

Determining the optimal set of transient states, as well as which action to take for all possible interweavings of messages, is a tedious and error-prone process when performed by hand. For this reason, there is a long-standing desire to develop

tools that automate this process and other combinatory problems that occur when developing complex, concurrent cache coherence protocols.

2.3.1 ProtoGen

One notable work, ProtoGen [10], introduces a generator capable of automatically producing fully concurrent cache coherence protocols from high-level stable state representations.

The authors developed a domain-specific language (DSL) that allows users to specify a cache coherence protocol's stable states and transitions. Based on this input, ProtoGen synthesizes the necessary transient states and transitions. During this process, ProtoGen automatically discovers opportunities for parallel execution and optimizes the concurrency of the generated protocol. By enabling the automatic generation of correct-by-construction protocols, ProtoGen can abstract away much of the complexity of protocol design.

The authors verified the correctness of various generated protocols using the Murφ model checker, ensuring both Single-Writer-Multiple-Reader (SWMR) properties and deadlock freedom were upheld. The code necessary for this verification was automatically generated, allowing for the easy verification of novel protocols.

However, ProtoGen is limited to generating flat protocols that only contain private caches and a single directory. This shortcoming limits its applicability to more complex systems, such as those with hierarchical or heterogeneous cache structures, as they require additional logic.

2.3.2 HieraGen

In their follow-up work, HieraGen [11], the authors addressed the limitations of ProtoGen by extending their approach to automatically generate hierarchical cache coherence protocols.

HieraGen builds upon the same input format used in ProtoGen but introduces the ability to generate memory hierarchies with multiple levels. This is achieved by automatically fusing the state machines of the directory and caches to form

a new, unified state machine that can act as a shared cache for multiple private caches. Moreover, HieraGen can further improve the concurrency of the generated protocol by optimizing state transitions for each level to minimize bottlenecks and improve overall system performance.

As with ProtoGen, the correctness of the generated protocols was verified using formal verification methods to ensure that the protocols remain deadlock-free and still adhere to memory consistency requirements. However, while HieraGen extends the framework to handle hierarchical cache systems, it remains limited to homogeneous environments where the protocols of all caches belong to the same family of coherence protocols.

2.3.3 HeteroGen

The design of a heterogeneous cache coherence protocol is a much more difficult task than that of a homogeneous protocol, as the joint protocol must faithfully preserve the semantics and constraints of both original protocols. Depending on the similarities between the two protocols being fused, this process significantly increases possible states, further increasing complexity over already complex homogeneous protocols.

While some attempts [12] at manually writing heterogeneous protocols were made, the general complexity of this problem makes doing so very unappealing. Additionally, heterogeneous systems are generally created by composing different kinds of homogeneous systems. Relying purely on handwritten protocols severely limits the flexibility of such a system, as each new composition would require a significant time investment to fuse the individual protocols correctly.

One attempt at automating this fusing process is HeteroGen [13]. HeteroGen also builds on ProtoGen for its input format and transient state capabilities. HeteroGen extends this base to allow for automatically synthesizing heterogeneous cache controllers by combining two distinct input protocols.

As the fusion process is fully automated, the risk of accidentally introducing inconsistencies in the intricate interactions between differing coherence models is significantly reduced. To formally verify this fact, HeteroGen extends the existing

Murφ model checks for deadlock-freedom by additional litmus tests to show strict adherence to the necessary memory consistency models.

2.4 Simulating Systems with gem5

While model checking can help with showing the correctness of novel cache coherence protocol, the performance of said protocol is another crucial factor. As performance heavily depends on the workload that is being executed, it is difficult to make purely theoretical claims about the performance of a cache coherence protocol. However, creating real prototype hardware is often extremely expensive and, therefore, not suitable for early exploration of the design space. System simulations can be used to bridge this gap and run practical benchmarks on highly experimental architectures.

One of the most popular tools for creating such simulations is *gem5* [14]. gem5 is a highly flexible, open-source, modular platform for computer architecture research and is widely used in both academia and industry research labs such as ARM Research, AMD Research, and Google. The base layer of gem5 is an event-driven simulation engine that sets the foundation for the various system components gem5 is able to simulate. gem5 is able to simulate CPUs with in- or out-of-order designs, GPUs [15], various memory families [16] such as DDR3/4 or HBM2, I/O devices, on-chip networks, and more. On top of that, the CPU models also support a wide range of Instruction Set Architectures (ISAs) like x86, ARM, RISC-V, PowerPC, SPARC, MIPS, and Alpha. As gem5 is made for exploring novel architectures, the design of the components is highly modular to allow users to easily extend, replace, or customize them.

However, this does not mean that modifying gem5 is a requirement for using it. For many use cases, it is sufficient to simply use the components already provided. To control which components are used and how they are connected, gem5 uses a Python script that is passed to the simulator via a command line parameter. gem5 executes this Python script from inside an embedded Python runtime that exposes all available system components as Python objects. These objects can be freely instantiated and expose various parameters that can be used

to customize them. Components can be connected by assigning their respective Python object to a matching property of another.

gem5 supports two kinds of simulation setups: Full-System (FS) mode and System-Call Emulation (SE) mode. In FS mode, gem5 will simulate the entire system, including the operating system (OS). In SE mode, gem5 will only run a user-level application inside the simulated environment by providing custom, gem5-specific implementations for all system calls the application makes. This mode is generally preferred for micro-architecture exploration, as the reduced scope significantly speeds up the simulation and reduces noise in performance measurements as system components such as the OS kernel are not captured. Despite the absence of an OS in the simulation, gem5 is able to run unmodified Linux binaries in SE mode.

For modeling cache coherence protocols, gem5 offers the Ruby subsystem. Ruby was originally developed as part of the GEMS project [17] before being combined with m5 [18] into gem5. In Ruby, the various components of a cache coherence system (caches, directories, etc.) are written in a domain-specific language called SLICC (Specification Language including Cache Coherence). SLICC allows the user to specify the states and transitions of the individual components in a more high-level manner than raw C++, which SLICC compiles down to. The flexibility of SLICC makes it easy to add a new cache coherence protocol to gem5 without having to modify gem5 internals. The gem5 repository already contains a series of premade Ruby cache coherence protocols that can serve as baselines against which to compare. These protocols include overly simplistic protocols focused on brevity, such as *MI-example* as well as reimplementations of real-world protocols, such as *MOESI-hammer* which was in AMD's Opteron chips.

The Garnet subsystem [19], [20] is responsible for realistically simulating the interconnect that connects the various cache coherence system components together. Garnet allows for the simulation of arbitrary topologies with the ability to individually specify the latencies on each link to account for hop delays, queuing delays, and routing overheads.

2.5 Non Uniform Memory Access

In many modern computing systems, not all memory can be accessed at the same speed due to inherent differences in memory architecture. In traditional NUMA (Non-Uniform Memory Access) systems, cores are divided into separate NUMA domains, each containing its own local memory bank. Cores within a domain can access their local memory with low latency and high speed. However, when a core needs to access memory from a different NUMA domain, the latency is significantly higher, resulting in slower access times and potential performance bottlenecks.

If the operating system is unaware of these memory access differences, its performance can vary widely depending on how memory is allocated across the NUMA domains [21]. One approach to mitigate this variance is interleaving the physical address space at the cache line level. This technique distributes memory accesses evenly across all memory locations, which can help balance the load. However, it may lead to suboptimal performance because no effort is made to leverage the relative importance of certain memory pages, particularly those that are accessed more frequently.

To address these challenges, most operating systems have become NUMA-aware and started explicitly managing memory allocation placement. These systems expose various memory management strategies to user-level applications through NUMA policies. A typical policy is to allocate fresh pages in the memory region that is closest to the core that triggered the allocation. If an application fails to specify a particular NUMA policy, the operating system typically defaults to a pre-configured allocation strategy.

However, NUMA systems are not the only architectures with uneven memory latencies. In disaggregated memory systems, memory is divided into fast, local memory, and slow, remote memory, typically connected via interfaces like CXL (Compute Express Link). The local memory operates at high speed and low latency, similar to traditional DRAM, while the remote memory resides on external memory expanders. Although this remote memory provides increased capacity, it has higher latency and slower access speeds.

While remote memory can be registered with the operating system as another NUMA domain, many existing NUMA policies do not apply effectively in this context, as remote memory tends to exhibit uniformly slower access for all cores. To better support this architecture, a new memory management strategy called *Memory Tiering* [22] has been proposed. The key concept behind Memory Tiering is to assign specific tiers to all memory regions in a system based on their access latency. When a new page is allocated, it is initially placed in the default tier and can then be transparently migrated to a higher or lower tier based on usage patterns. Others propose a modified allocator [23] that allows users to explicitly request a specific memory tier during memory allocation.

An alternative method involves not directly exposing remote memory as system memory but instead presenting it as a block device [24]. In this case, it becomes the user's responsibility to correctly map the block device's address space into the current process's virtual address space. This method can offer the most control over how remote memory is utilized, though it requires greater programming effort and awareness of the underlying memory architecture.

3 Motivation

Modern systems are becoming increasingly heterogeneous, with various types of processors working together. Ensuring the correctness of cache coherence in such systems is a complex challenge. To avoid dealing with this complexity, data sharing between processors is often severely restricted, negatively impacting performance and programming flexibility. Improved simulations of heterogeneous systems can provide hardware vendors with the confidence needed to increase memory sharing between processors. However, setting up these simulations is a laborious task, often requiring users to manually replicate cache coherence protocols in a format compatible with the simulator. This manual process is not only time-consuming but also prone to errors and potentially results in a simulation that does not accurately reflect real-world behavior. Our goal is to address these issues by automating the process of creating simulations for heterogeneous systems, reducing the potential for mistakes, and streamlining the overall workflow.

4 Design

4.1 Architecture of a Cluster with Disaggregated Memory

The heterogeneous system we want to simulate in this work is a multi-node cluster with disaggregated memory (see Figure 3). The system consists of several compute nodes with different architectures, each containing multiple cores. However, rather than each node having its own local memory and exchanging data using traditional message-passing techniques, the majority of the system's memory resides in a dedicated memory node. This memory is shared with the compute nodes via a high-speed cache-coherent interconnect network.

This setup allows the compute nodes to communicate using conventional shared-memory programming techniques, while reducing memory overhead by avoiding duplication of shared data structures. Centralizing memory also serves to allow for more efficient resource distribution.

In practice, most nodes will still have a small amount of private, local memory to avoid the performance costs of frequently accessing data over the interconnect network. However, for the purposes of this work, we primarily focus on the shared memory aspects of the system and often omit the private memory in our simulations. The exception is Section 5.2, which explores how private memory can be incorporated into the simulation.

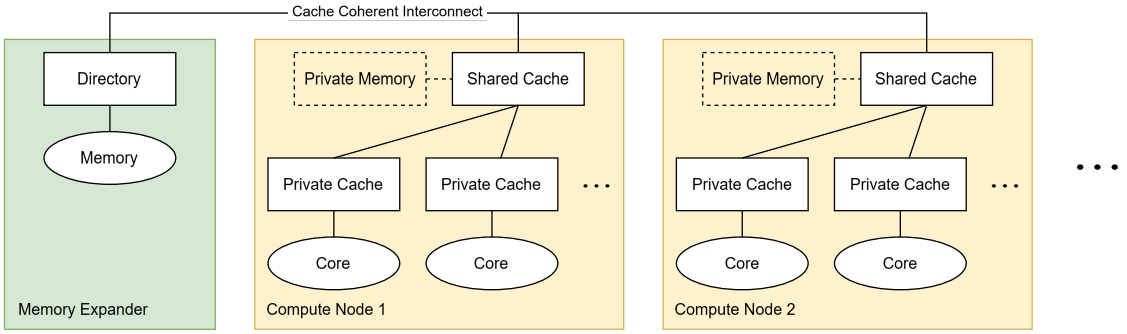


Figure 3: Diagram of this works target system architecture.

4.2 Heterogeneity in the System architecture

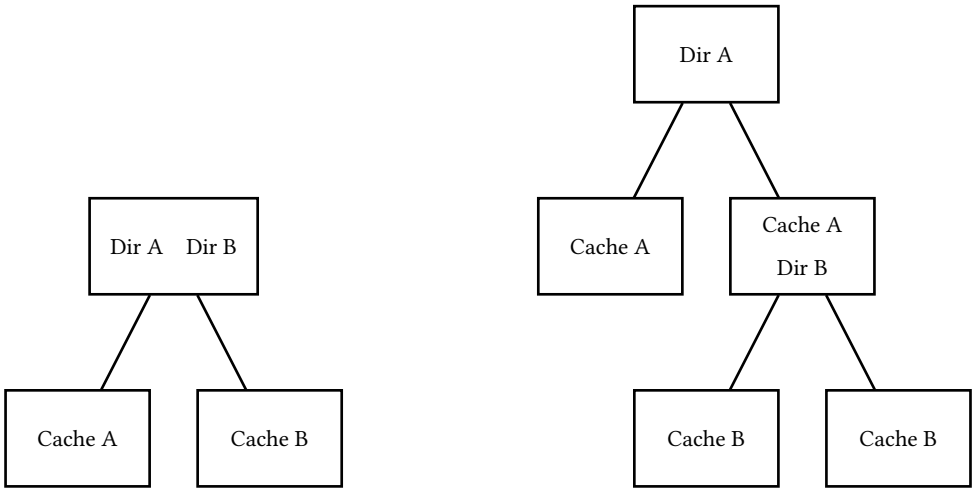
A key aspect of the system design outlined in Section 4.1 is that the compute nodes can have different architectures, each potentially using a different cache coherence protocol. To enable these heterogeneous nodes to share the same memory, at least one system component must be capable of supporting multiple protocols. This section explores two strategies for enabling multiple cache coherence protocols to coexist in a single system: Fused Directories and Bridges.

4.2.1 Heterogeneity via Fused Directory

The first strategy is to fuse all directory implementation of all protocols in the system in to a big unified directory, allowing it to handle coherence request from all compute nodes (Figure 4a).

In a fused directory, the states are typically derived as the cross-product of the states from the source directories. This enables the directory to independently track the coherence states of both protocols and respond to each cluster in accordance with its respective protocol. However, this arrangement does not inherently guarantee coherence between the two clusters. For instance, a state combination like M+S would lead to an incoherent scenario, as the M (Modified) state implies exclusive ownership and can not coexist with the S (Shared) state on the other side.

To resolve this, the state transitions on one side must be adjusted to trigger corresponding transitions on the other side, ensuring that any conflicting or incoherent access rights are invalidated. If the directory is in the I+S state, for example, and the first cluster requests exclusive access (M), the second cluster must first be downgraded to I. As a result, the final state of this transaction becomes M+I rather than the incoherent M+S state. By consistently applying this strategy many states become unreachable and can be pruned from the final state machine. Nevertheless, the resulting state machine generally has significantly more states than either of the source state machines. This complexity growth puts significant limits on the extensibility of this approach when expanding to a system with more than two protocols.



(a) Fused Directory (b) Bridge

Figure 4: A comparison between the *Fused Directory* and *Bridge* architectures.

Additionally, integrating a new protocol into an existing system requires modifications to the existing directory design. In real-world systems, such changes would necessitate costly and time-consuming redevelopment of hardware, posing a significant barrier to the integration of nodes with novel architectures.

4.2.2 Heterogeneity via Bridge

Another approach to creating a heterogeneous cache coherence system is to establish a hierarchy between the two source protocols. In this design, instead of using a unified directory shared across all protocols, the directory remains exclusively under the control of one protocol, referred to as the *parent protocol*. To enable other protocols (or *child protocols*) to access the shared memory region, a bridge is introduced, which acts as an intermediary that translates coherence requests between protocols (Figure 4b).

From the perspective of the child protocol, the bridge assumes the role of a directory, handling coherence requests and maintaining the coherence state of the cluster. Conversely, from the viewpoint of the parent protocol, the bridge behaves like any other cache, storing copies of cache lines and generating coherence requests when more are needed. One advantage of this hierarchical structure is that the existing system requires minimal redesign when adding bridges. Since the bridges behaves like a standard cache from the parent protocol's perspective, no

fundamental changes to the coherence protocol are needed, making this approach more modular and scalable than the fused directory model. Conceptually, bridges replace the shared caches shown in Figure 3 but can also be added as separate component between the shared cache and the directory.

The process of creating a proxy cache is conceptually similar to creating a fused directory, but with a key distinction: instead of merging two directories, a directory and a cache are combined. This fusion impacts how the two controllers, one for the directory and one for the cache, must be coordinated. In a proxy cache, the directory side must respect the access permissions of the cache side as the cache conceptually replaces the memory controller. For instance, if the directory grants write permissions to one of its subordinate caches, the cache side of the proxy must first secure the necessary write permissions for itself before granting them further downstream. This hierarchical enforcement ensures coherence is maintained between the protocols while avoiding the complexity of directly integrating multiple directories.

4.3 Automatic Synthesis of Required Protocols

As our goal is to make constructing heterogeneous simulations as seamless as possible, we aim to specify the internal protocols of each node in our system at a high level while offloading the composition of said protocols to an automated process. This automation allows for scalability, as new nodes or architectures can be incorporated into the simulation without requiring manual modifications to the existing protocol infrastructure.

4.3.1 Protocol Specification for Nodes

A major decision in the scope of this work was how to specify the internal cache coherence protocols for individual nodes. One option was to assign each node a Ruby protocol written in SLICC. The advantage of this approach is that gem5 already provides several protocols in its standard library, which could be reused for our purposes. Additionally, these protocols are natively supported by gem5 and do not require any transpiling before they can be used.

However, a significant drawback of using SLICC as the specification format is its relatively low-level nature, making the development of new protocols more complex, especially for architectures that do not have suitable options in gem5′s standard library. Another limitation of SLICC is its specificity to gem5, which impedes the use of these protocols in alternative verification methods, such as model checking, which could provide additional confidence in the correctness of the protocols produced by our system.

For these reasons, we chose to use the format introduced by ProtoGen [10] as the input format for our system. This format allows cache coherence protocols to be expressed purely through their stable state representations, with ProtoGen automatically generating the necessary transient states. As a result, the protocol description files are significantly simpler than their SLICC equivalents. ProtoGen also supports formal verification of deadlock freedom using the Murφ model checker. Most importantly, there is existing work on generating heterogeneous protocols from ProtoGen descriptions (HeteroGen [13]), making it an ideal choice for the basis our simulation framework.

4.3.2 Protocol Generation from High Level Descriptions

The bridge generator is fundamentally the result of combining the concepts from HieraGen [11] and HeteroGen [13]. While HeteroGen enables the fusion of two protocols to create a fused directory, as described in Section 4.2.1, HieraGen supports the transformation of a flat protocol into a hierarchical one. The bridge generator merges these two ideas, allowing two different flat protocols to be joined into a heterogeneous, hierarchical protocol, as described in Section 4.2.2.

In our case, we fuse the individual protocols used by the compute nodes with the protocol of the memory expander, creating a bridge component for each compute node. Additionally, we generate complete protocol definitions (including transient states) for the other system components that remain homogeneous.

4.3.3 Integration with gem5

To integrate our generated protocols into gem5, we introduce a custom compiler backend that automatically produces a complete Ruby cache coherence protocol in

SLICC. The generated protocol contains all controllers for all components required for the simulation, such caches and bridges for the compute nodes and a directory for the memory expander. As SLICC is a human-readable format, the resulting controllers can still be manually inspected before compute compiling them into gem5 or during troubleshooting. Once compiled the protocol can be reused for multiple different system configuration, only requiring regeneration when another architecture is added to the system.

4.4 Reusable Simulator Setup

We developed a flexible setup script that allows dynamic configuration of various simulation parameters, such as the number of compute nodes, the number of cores per node, and the architecture of each node. Additionally, the communication latencies within the system can be adjusted to accurately model the overhead of accessing memory from the memory expander. This script is designed for reusability, ensuring that the generated protocols can seamlessly integrate into the simulation environment without requiring extensive manual adjustments.

By default, all memory in the simulated system resides within the memory expander. However, we also support an alternative configuration where nodes use local memory by default, and memory must be explicitly allocated on the memory expander using a custom allocator. The complete simulation setup is capable of running unmodified Linux binaries and testing system performance using standard benchmark programs.

5 Implementation

5.1 Configuration of gem5

The configuration of gem5 is done by passing a Python script as an input parameter. This script is processed by an embedded Python runtime within gem5. Users must first instantiate all the necessary simulation components within the script before initiating the simulation.

5.1.1 Cores and Memory

The core of the configuration script is the `system` object, to which all other components, such as CPU core or Memory, are attached. Internally, these Python objects are automatically generated wrappers around the C++ classes gem5 uses to model system components.

gem5 provides two different memory modes to accommodate various use cases: *Timing* and *Atomic*. The Timing mode is more accurate, simulating queuing delays and resource contention, while the Atomic mode is faster but less precise.

```
import math
import m5
from m5.objects import *

system = System()

system.clk_domain = SrcClockDomain()
system.clk_domain.clock = "1GHz"
system.clk_domain.voltage_domain = VoltageDomain()

system.mem_mode = "timing"
system.mem_ranges = [AddrRange("1GB")]
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]

system.cpu = [X86TimingSimpleCPU(cpu_id=i) in i for range(2)]
for cpu in system.cpu:
  cpu.createInterruptController()
```

Listing 1: Instantiation of System, CPUs, and Memory for a gem5 simulation setup.

The atomic mode also supports some additional features, such as simulation forwarding. Technically, there exists a third memory access mode in gem5 called the *Functional* mode. However, this mode is mainly used for debugging rather than general use. Section 5.6.1 contains a more detailed description of this access mode.

The CPUs used must align with the chosen memory mode. CPU objects follow a naming convention of `{ISA}{mem_mode}{cpu_type}`, where `ISA` and `mem_mode` are straightforward, and `cpu_type` refers to the specific CPU model. The setup shown in Listing 1 uses the `SimpleCPU` model. A more complex alternative to this model is the out-of-order (`O3CPU`) model.

5.1.2 Ruby Caches

There are two different cache subsystems in gem5. Simple caches are fast to simulate but inaccurate, while Ruby caches are more accurate at the expense of being slower to simulate and more complex in their use. As this work centers around the simulation of cache coherence protocols, we have to use Ruby. As Ruby is centered around components and their communication with each other, it is paramount that each component has a unique identity so that different components can address each other. This identity is formed from the type of a controller (e.g., `L1Cache`) and a version. Caches are connected to their respective core via a `RubySequencer`. The sequencer is responsible for turning load and store requests from the CPU into messages that the cache controller can process. All Ruby controllers communicate with each other by sending messages to each other. For this reason, an important part of configuring controllers are the message queues. Contrary to what one might expect, the corresponding endpoints are not connected directly, but instead, each endpoint gets its own `MessageBuffer`, and it is the network's (Section 5.1.3) job to move messages between buffers.

```
ruby_system = RubySystem()
ruby_system.sys_port_proxy = RubyPortProxy()
system.system_port = ruby_system.sys_port_proxy.in_ports
system.caches = ruby_system

sequencers = []
controllers = []

cache_size_bits = int(math.log(system.cache_line_size, 2))
for cpu in system.cpu:
  l1cache = L1Cache_Controller()
  l1cache.version = cpu.cpu_id # unique id per type
  l1cache.cacheMemory = RubyCache(
    size = '16kB', assoc = 8, start_index_bit = cache_size_bits)
  # ...
  self.mandatoryQueue = MessageBuffer()
  self.requestFromCache = MessageBuffer(ordered=True)
  self.requestFromCache.out_port = ruby_system.network.in_port
  # ...

  sequencer = RubySequencer(version=cpu.cpu_id, dcache=l1cache.cache, ...)
  l1cache.sequencer = sequencer
  sequencer.connectCpuPorts(cpu)

  sequencers.append(sequencer)
  controllers.append(l1cache)

ruby_system.num_of_sequencers = len(sequencers)

dir = Directory_Controller()
dir.version = 0
self.addr_ranges = sytem.mem_ranges[0]
self.directory = RubyDirectoryMemory()
self.memory_out_port = system.mem_ctrl
# ...
self.requestToMemory = MessageBuffer()
self.responseFromMemory = MessageBuffer()
self.requestToDir = MessageBuffer(ordered=True)
self.requestToDir.in_port = ruby_system.network.out_port
# ...
controllers.append(dir)
```

Listing 2: Instantiation of the Ruby cache system for a gem5 simulation setup.

```
network = GarnetNetwork()

ruby_system.network = network
ruby_system.number_of_virtual_networks = 3
ruby_system.network.number_of_virtual_networks = 3

network.routers = [GarnetRouter(router_id=i) for i in range(len(controllers))]
network.ext_links = [
  GarnetExtLink(link_id=i, ext_node=c, int_node=network.routers[i], latency=8)
  for i, c in enumerate(controllers)
]
network.netifs = [GarnetNetworkInterface(id=i) for i in
range(len(network.ext_links))]

int_links = []
for ri in network.routers:
  for rj in network.routers:
    if ri == rj:
        continue  # Don't connect a router to itself!
    self.int_links.append(
      GarnetIntLink(link_id=len(int_links), src_node=ri, dst_node=rj, latency=8))
network.int_links = int_links
```

Listing 3: Instantiation of the Garnet network for a gem5 simulation setup.

5.1.3 Garnet Interconnect

gem5 provides two models for network simulation, each balancing accuracy and simulation speed. Since we focus on evaluating the impact of connecting multiple processors to an external memory bank, obtaining precise performance metrics for the network is critical. Therefore, we opt for the more accurate `GarnetNetwork` model rather than the faster but less precise `SimpleNetwork` model. The key component in this network is the `GarnetRouter`, which connects to other routers via internal links and to system components via external links. This minimalistic design supports the replication of common network topologies, such as point-to-point, grid, crossbar, or any other graph structure. Internally, the network is divided into virtual networks that ensure that each message is only visible within its assigned virtual network. This structure allows controllers to manage which messages appear in which specific queues.

```
binary = 'hello-world'
system.workload = SEWorkload.init_compatible(binary)

process = Process()
process.cmd = binary

for cpu in system.cpu:
  cpu.workload = process
  cpu.createThreads()

root = Root(full_system=False, system=system)
m5.instantiate()

print(f"Beginning simulation!")
exit_event = m5.simulate()
print(f"Exiting @ tick {m5.curTick()} because {exit_event.getCause()}")
```

Listing 4: Instantiation of the executed binary for gem5 simulation setup.

5.1.4 Simulated Workload

The final step in the simulation setup is to define the workload by creating a `Process` object. This object is also used to configure options such as input/output redirection and command line arguments for the target binary. Afterward, the process must be assigned to individual cores. It is crucial that the number of threads spawned by the process does not exceed the number of available cores in the simulation. Since the SE mode lacks a scheduler, exceeding the core count typically results in a crash, as newly spawned threads will fail to start as expected. Once the simulator is instantiated, the simulation can be started using the `m5.simulate()` function. This function returns when an event is triggered during execution. While this usually occurs when the simulated process completes its execution, gem5 offers custom instructions to manually trigger additional events that can be compiled into the workload binary. However, since we do not use these for our project, we can assume that a single call to simulate will run the simulation to completion.

5.1.5 Execution Results

To observe the execution of the workload, gem5 offers several debug flags such as `ProtocolTrace` which enables logging of every state machine transition in the Ruby cache system. As these transitions happen very frequently, enabling this debug flag significantly slows down the simulation. After a successful run, gem5 additionally write a `stats.txt` file that contains a details report of all performance counters recorded during the simulation. These recorded stats include:

- The amount of time that was simulated
- The number of instructions executed
- The number of memory reads and writes
- The number of cache hit and misses for each controller
- The number of messages that passed through each queue

5.2 Supporting NUMA Awareness

To run benchmarks that specifically test disaggregated memory, we need to replicate NUMA strategies within the gem5 simulator. However, in System-Call Emulation (SE) mode, gem5 lacks access to most operating system functionality, which heavily limits the NUMA techniques outlined in Section 2.5 we can use. Both classic NUMA policies and memory tiering heavily rely on the operating system to work and, therefore, do not work in SE mode. As a result, the only strategies we could replicate in SE mode are physical address interleaving and explicit memory mapping.

5.2.1 Physical Address Interleaving

When building the physical address space, this strategy continuously changes between local and remote memory with cache line size granularity, causing the load to be evenly distributed between both memory regions. This causes the strategy to not be particularly useful for benchmarking purposes as the distribution does not follow any sensible pattern, such as usage frequency. However, it is helpful for debugging and testing the correctness of the system as it is an easy way to ensure that both memory regions are actively in use. This is made all the more

valuable because this strategy does not require any changes to the benchmark's source code.

Implementing an interleaved memory strategy in gem5 is relatively straightforward, as gem5 includes built-in support for interleaved address ranges. Each memory controller's `dram.range` field must be assigned an `AddrRange` with the appropriate interleave settings (using the `intlv*` fields) to use this feature. Additionally, the address ranges for all memory controllers need to be combined into a single, contiguous address space in the `system.mem_ranges` list. This means the individual address ranges managed by each memory controller must be manually combined into a unified address range of the entire system.

5.2.2 Explicitly Mapped Memory

Memory connected over CXL can be configured so that it is not automatically added to the system memory pool. Instead, the memory is exposed as a block device. If a program wants to utilize this additional memory, it must first manually map it using a function like `mmap`. This function lets a program map a range of a selected block device to an address in its virtual address space. In practice, most programs allow the operating system to pick the address in the virtual address space to which the memory is mapped. Afterward, the mapped region can either be used directly or used as a memory pool for a custom allocator and dynamically subdivided into smaller chunks as required.

Unfortunately, this behavior can, to our knowledge, not be directly replicated in gem5′s SE mode as simulating I/O devices requires the Full System (FS) simulation mode. However, gem5 exposes several functions to directly manipulate the virtual address space of the simulated program to the Python configuration script. One of these functions can directly create a new physical-to-virtual address mapping that closely mirrors the aspects of `mmap` required to implement this memory management strategy. A key difference is that the mapping process occurs in the configuration script, not the program itself. Consequently, calls to `mmap` in the benchmark program must be manually replaced with the hard-coded addresses,

```
target_virtual_address = 0x10000000
#...
process = Process()
#...
root = Root(full_system = False, system = system)
m5.instantiate()

process.map(target_virtual_address,
  remote_addr_range.start, remote_addr_range.size())

exit_event = m5.simulate()
#...
```

Listing 5: Manually mapping memory into the address space of a simulated program.

where the remote memory was mapped before the program can be used in the simulation.

Listing 5 demonstrates the changes required to the configuration script to set up a manually mapped memory region. `remote_addr_range` is the address range of a second memory controller that is not part of the system memory range.

5.3 Custom Cache Coherence Protocols with Ruby

Cache controllers for the Ruby memory subsystem are written in the domain-specific language SLICC. When building gem5, the SLICC compiler processes the SLICC files and generates C++ classes, which are then compiled into the gem5 binary itself. A consequence of this workflow is that any changes to SLICC files require gem5 to be recompiled. Protocols are also mutually exclusive, and at most, one protocol can be active at once. It, therefore, is common practice to use multiple different build configurations in parallel when comparing different protocols. However, C++ is not the only compilation target of SLICC. If desired, the SLICC compiler can also output the transition tables of the protocol as HTML files. For this reason, most syntactic constructs in SLICC have fields to supply human-readable names or descriptions.

```
enumeration(CoherenceRequestType, desc="Types of request messages") {
    GetM,       desc="Request from cache for a block with write permission";
    //...
}

structure(RequestMsg, desc="Used for Cache->Dir and Fwd messages",
interface="Message") {
    Addr addr,                    desc="Physical address for this request";
    CoherenceRequestType Type,    desc="Type of request";
    MachineID Requestor,          desc="Node who initiated the request";
    NetDest Destination,          desc="Multicast destination mask";
    DataBlock DataBlk,            desc="data for the cache line";
    MessageSizeType MessageSize, desc="size category of the message";

    //ommited: functional access functions
}
```

Listing 6: The message definition for a Ruby protocol.

5.3.1 Defining Messages

A cache system contains multiple components that communicate with each other by sending and receiving messages with predefined types. In gem5, all messages sent on the same virtual channel must be of the same type. To nevertheless be able to send different kinds of messages it is common practice to first define an enumeration (Listing 7) to differentiate between different kinds of messages and indicate how the remaining fields should be interpreted. The actual message type is a simple structure containing this enumeration and other required information, such as the sender, destination, or target cache line address. Some fields, such as the message size, are mandatory, but most can be freely chosen.

5.3.2 Declaring Components

The entry point for a Ruby cache controller in gem5 is the machine declaration (Listing 7). Each machine has a unique machine type that defines the name of the generated controller. For instance, the machine type `MachineType:L1Cache` will generate a controller named `L1Cache_Controller`, which can then be referenced in the configuration script. This machine type must also be unique across the network, as it is combined with a version ID to form the address of each controller.

```
machine(MachineType:L1Cache, "MI cache")
: Sequencer *sequencer; CacheMemory *cacheMemory; bool send_evictions;

  MessageBuffer * mandatoryQueue;
  MessageBuffer * requestFromCache, network="To", virtual_network="0",
vnet_type="request";
  //...
{
  state_declaration(State, desc="Cache states") {
    I,  AccessPermission:Invalid, desc="Not present/Invalid";
    M,  AccessPermission:Read_Write,
            desc="Modified. Read & write permissions. Owner of block";
    IM, AccessPermission:Busy,  desc="Waiting for GetM response";
    //...
  }
  enumeration(Event, desc="Cache events") {
    Load,  desc="Load from processor";
    Store, desc="Store from processor";
    //...
  }
  //ommited:
  // - functional access functions
  // - interal structure definitions
  // - function prototypes
  // - helper functions
  out_port(requestNetwork_out, RequestMsg, requestFromCache);
  //...
  in_port(mandatoryQueue_in, RubyRequest, mandatoryQueue, desc="...") {
    // Future section
  }
  //...
  action(a_issueRequest, "a", desc="Issue a request") {
    // Future section
  }
  action(m_popMandatoryQueue, "m", desc="Pop the mandatory request queue") {
    // Future section
  }
  //...
  transition(I, Store, IM) {
    a_issueRequest; m_popMandatoryQueue; //...
  }
  //...
}
```

Listing 7: The definition of a Ruby cache controller.

In addition to the machine type, machines have various parameters, such as message queues for communication with other controllers or references to allocated cache memory. These parameters must be initialized in the configuration script if they do not have default values. As the primary purpose of the parameters is to allow for dynamic configuration of the cache controller from the configuration script, there are almost no restrictions on numbers, names, or data types of parameters. Message queues require some additional specifications, such as the virtual network number. The virtual network number controls which message queues will connect to one another.

A special case is the `mandatoryQueue`, which is used by the Sequencer to send memory requests to the cache controller. This queue must have this exact, predefined name, or gem5 will not recognize it. However, the `mandatoryQueue` is only required for controllers attached to a Sequencer, such as private caches. Shared caches and directories, which do not have their own Sequencers, do not require this queue.

The different states of a machine are declared within the `state_declaration` block. These states themself are a central part of the cache coherence protocol being implemented to the point that protocols are commonly named after their primary states. Each state has a name and an access permission. While the controller itself does not directly use these access permissions, they are essential for gem5′s functional access system (Section 5.6.1).

Another key element in the machine declaration is the specification of events the controller will handle. Events, which are triggered by the `in_ports`, cause the controller to perform state transitions. These transitions are defined in `transition` blocks, each of which consists of four parts: the start state(s), an event, an end state, and a list of actions. If the specified event occurs while the machine is in one of the start states, the controller will execute the actions and then transition to the end state. If the end state is not provided, the machine will remain in its original state after executing the actions. It is important to note that the SLICC compiler will generate an error if any start state and event combination is not unique. While

missing combinations will not cause errors during compilation, they will cause a simulation crash if encountered during runtime, resulting in an “invalid transition” error.

5.3.3 Sending And Receiving Messages

To use a message queue from the controller, it first needs to be bound to a name (Listing 8). This happens in the `out_port` statement for outgoing queues and the

```
out_port(requestNetwork_out, RequestMsg, requestFromCache);

in_port(mandatory_in, RubyRequest, mandatoryQueue) {
  if (mandatory_in.isReady(clockEdge())) {
    peek(mandatory_in, RubyRequest, block_on="LineAddress") {
      Entry cache_entry := getCacheEntry(in_msg.LineAddress);
      TBE tbe := TBEs[in_msg.LineAddress];
      //omitted: cache replacement
      if (in_msg.Type == RubyRequestType:LD ||
            in_msg.Type == RubyRequestType:IFETCH) {
        trigger(Event:Load, in_msg.LineAddress, cache_entry, tbe);
      } else if (in_msg.Type == RubyRequestType:ST) {
        trigger(Event:Store, in_msg.LineAddress, cache_entry, tbe);
      } else {
        error("Unexpected type from processor");
      }
    }
  }
}

action(a_issueRequest, "a", desc="Issue a request") {
  enqueue(requestNetwork_out, RequestMsg, issue_latency) {
    out_msg.addr := address;
    out_msg.Type := CoherenceRequestType:GetM;
    out_msg.Requestor := machineID;
    out_msg.Destination.add(mapAddressToMachine(address, MachineType:Directory));
    out_msg.MessageSize := MessageSizeType:Control;
  }
}
action(m_popMandatoryQueue, "m", desc="Pop the mandatory request queue") {
  mandatoryQueue_in.dequeue(clockEdge());
}
```

Listing 8: The message handling of a Ruby cache controller.

in_port statement for incoming queues. In this case, the write side of the queue requestsFromCache will be bound to the name requestNetwork_out. The third parameter is the message type. In the case of the out_port, this is the message defined in Listing 6. In the case of the in_port, the message type is RubyRequest as we are connecting to the special mandatoryQueue and therefore must use the same message type as the Sequencer.

When reacting to incoming messages, the controller will continuously execute all in_port blocks in sequential order until all messages are processed. As in_port blocks are called sequentially, the in_port blocks should be ordered by priority. Another consequence of this design is that in_port blocks can be called even if no new message is available for them. For this reason, the first action inside of an in_port block is usually to check whether a processable message is available before attempting to read it. The reading itself is done using the peek function. Inside the scope attached to this function, the peeked message can be accessed using the name in_msg. By looking at the type enumeration of the message we can translate the incoming message to the corresponding event. Once an event is triggered, the current processing attempt will abort, and a new one will start from the top. If no in_port triggered an event, the controller will go to sleep until it is woken up again by a new message. In general, there is a close mapping between the type of incoming messages and events. However, in some cases, some additional control flow might be necessary to determine the correct event. It is also important that all branching, such as acknowledgment counting, is done in the in_port as gem5 assumes that actions always follow the same code path.

Sending messages works every similar similarly to reading messages except that the special variable inside the enqueue scope is called out_msg instead of in_msg. If the message being sent is a response, the destination is typically the sender of the request (in_msg.Sender). If the message is a request, the destination must often be dynamically determined. For more information on this process, see Section 5.6.3.

```
machine(MachineType:Directory, "Directory protocol")
: DirectoryMemory * directory;
  Cycles to_memory_controller_latency := 1;

  MessageBuffer * requestToDir, network="From", virtual_network="0",
                                                    vnet_type="request"
  //..
  MessageBuffer * requestToMemory;
  MessageBuffer * responseFromMemory;
{
  state_declaration(State, desc="Directory states", default="Directory_State_I") {
    I, AccessPermission:Read_Write, desc="I for all caches";
    M, AccessPermission:Maybe_Stale, desc="M in one cache";
    IM, AccessPermission:Busy, desc="Moving to M but waiting for memory";
    //...
  }
  enumeration(Event, desc="Directory events") {
    GetM, desc="Cache requests write permissions";
    MemoryData, desc="Fetched data from memory arrives";
    //...
  }
  out_port(memQueue_out, MemoryMsg, requestToMemory);
  //...
  in_port(requestQueue_in, RequestMsg, requestToDir) {
    // trigger(Event:GetM)
  }
  in_port(memQueue_in, MemoryMsg, responseFromMemory) {
    // trigger(Event:MemoryData)
  }
  // transition(I, GetM, IM) -> send request to memory
  // transition(IM, MemoryData, M) -> send data to cache

  //omitted: all other components already outlined in the cache controller
}
```

Listing 9: The implementation of Ruby directory controller.

5.3.4 Directory Implementation

The component demonstrated in the previous sections is a controller for private cache. It represents the lowest level of the cache hierarchy and the touch point between the custom protocol and the simulated CPU cores. The other end is a directory controller, representing the highest level of the cache hierarchy and the

touch point between the custom protocol and the memory controller. Between these two mandatory controllers, there can be an arbitrary number of additional components such as an shared cache. These generally only have to interact with other custom components and are, therefore, almost entirely unconstrained in purpose and behavior.

Conceptually, the directory controller is very similar to the private cache controller. The main difference is that instead of having a special queue for connecting to a Sequencer, it has a pair of special queues for connecting to a memory controller. Another oddity is that the functional access permissions of the states seem to be reversed. The reason for this is that the access permissions follow a cache-centric perspective. A cache line being in the *M* state means that one cache currently holds a writable copy of that cache line, and any changes to the directory's version of the same cache line will be overwritten when the cache performs a write back.

5.4 Simulating Heterogeneous Systems

Simulating heterogeneous cache coherence systems inherently requires multiple cache coherence protocols. This presents a challenge in gem5, as the cache coherence protocol is chosen at compile time, and therefore, only one protocol can be active for each simulation run.

This section details the strategies for overcoming this limitation and outlines the additional changes required to simulate heterogeneous cache coherence systems in gem5 accurately.

5.4.1 Extending Machine IDs

The obvious solution to being unable to compile gem5 with more than one protocol active at the same time is simply to create a new combined protocol that includes all the controller and message structures defined by the source protocols and adapt the source code to solve name collisions. This process is fairly straightforward for messages as they can be freely renamed. Machines, however, do not have an inherent name but instead derive their name from their machine

type. As machine types must be unique, having two machines with the machine type `MachineType:L1Cache`, for example, is impossible. To solve this, the default `MachineType` enumeration provided by gem5 must be extended. An easy way to fix the collision is to split `L1Cache` into `L1CacheA` and `L1CacheB`. This is possible because most of the entries of this enumeration are simply treated as opaque values by gem5 and not directly referenced in the source code, so modifying this enumeration does not cause any issues in practice. With this change in place, the two distinct types of L1 cache controllers can now be accessed using the names `L1CacheA_Controller` and `L1CacheB_Controller` respectively from the Python configuration script and used in tandem. Using these techniques, it becomes possible to use two different cache coherence protocols simultaneously in the same gem5 build.

5.4.2 Merging Messages

While it is relatively straightforward to rename message structures to allow them to compile side-by-side, working with multiple distinct message types is not ideal. The main issue is that each virtual channel can only carry one message type. As a result, the virtual channel IDs of the second protocol must be offset by the number of virtual channels used by the first protocol. However, assigning different virtual channels to different message types also means that any heterogeneous component interested in messages from multiple protocols requires many additional `in_port` handlers, which significantly increases the implementation complexity of the component.

To improve interoperability between protocols, it is beneficial to consolidate the different message types and kinds into shared message structures. To ensure that messages can still be correctly identified by their originating protocol and to avoid naming collisions, members of the shared message kind enumeration can be prefixed with the name of the protocol to which they belong.

5.4.3 Communication Patterns for Bridges

A key challenge when merging two existing protocols is ensuring that components using different protocols communicate only through a bridge where messages can be translated appropriately. Handling requests to the directory is relatively

straightforward: instead of sending the request directly to the directory, it is routed to the bridge. The bridge then translates the request and forwards it to the appropriate directory. Notably, the bridge must modify the sender field of the request to itself, preventing the directory from replying directly to the originating cache. Instead, the directory's response must return to the bridge, which translates the response back into a format the requesting cache can interpret.

One important consideration with this approach is that the directory will now only recognize the bridge as the contact point for coherence information. As a result, the sharer list maintained by the directory will reference the bridge instead of the individual caches. This requires the ownership states of private caches to be tracked either below the bridge or within the bridge itself, ensuring that write-back requests are correctly propagated.

While these vertical flows (from cache to directory and back) are relatively easy to manage, many protocols also require lateral communication between caches. For example, invalidation acknowledgments must often be counted by the requesting cache rather than the directory. In such cases, caches that have completed an invalidation send acknowledgments directly to the requesting cache. However, when bridges are introduced, caches may belong to different protocol regions, complicating this direct communication.

Sending the acknowledgment directly to the bridge responsible for the original request would be incorrect, as it may not conform to the proper protocol. Instead, the acknowledgment must first be sent to the sender's local bridge, which then forwards it to the requestor's bridge. This second bridge then translates and forwards the acknowledgment to the original requesting cache. This ensures that all caches only receive the appropriate message types.

5.4.4 Clustered System Configuration

A core characteristic of the cluster we simulate is that intra-node communication is faster than inter-node communication, including communication with the memory node. To replicate this accurately in our simulation, we use different link latencies between various components when configuring our Garnet network (see Figure 5).

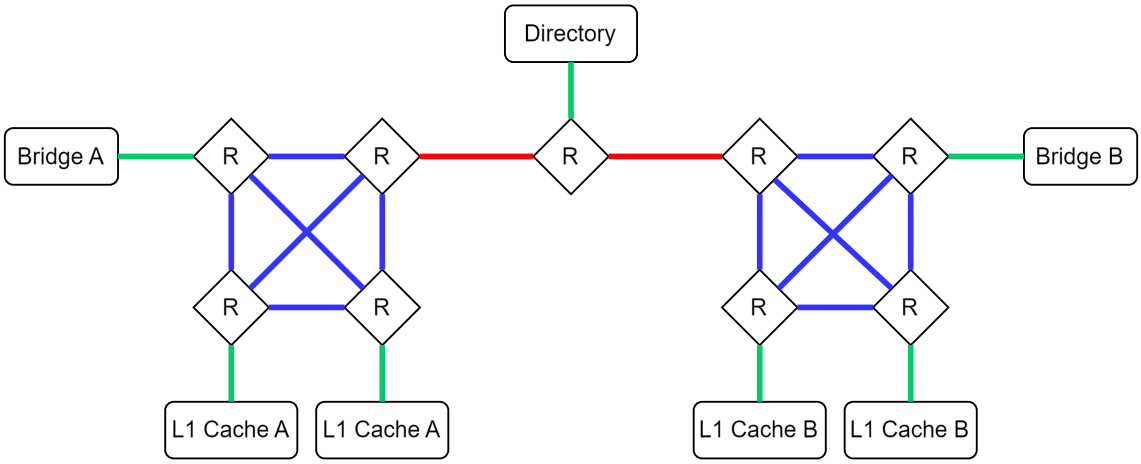


Figure 5: Diagram of network architecture in our heterogenous simulation setup. Latency-Ordering: Component (Green) < Local (Blue) < Remote (Red)

For external connections (*Component*), which link Ruby controllers and Garnet routers, we assume negligible latency, as these components are assumed to be co-located. Intra-node communication (*Local*) is modeled using a low-latency, point-to-point network, reflecting the faster communication within each node. For inter-node communication (*Remote*), we simulate a star topology with higher latency, with the directory located at the center to model the slower communication between nodes and the memory node. Critically, communications between two compute nodes must cross the slow *Remote* link twice.

When configuring remote memory as external memory, as described in Section 5.2, we introduce a second directory to the simulation. However, instead of using *Remote* latencies, we apply *Local* latencies to connect this directory to the individual clusters. It is important to note that while this directory uses local latencies, it is still shared between the clusters and does not represent genuinely local memory.

To ensure accurate benchmark results in this configuration, it is critical that the benchmarks rely solely on external memory for data sharing and avoid traditional shared memory programming techniques. This prevents skewed results caused by unintended use of shared memory between the individual clusters.

5.5 Ruby Protocol Generation

Manually building each bridge contradicts our aim of providing a solution that incurs minimal friction when composing novel heterogeneous architectures. To address this issue, we automated the process of bridge creation. As cache coherence protocols for gem5 are written in SLICC, we must automatically emit SLICC code to utilize these generated bridges in our simulation.

As the SLICC generation builds on the bridge generation, Section 5.5.1 will first introduce the bridge generator by discussing its input and output in the context of turning its output into valid SLICC code. The following sections will then discuss the process of SLICC generation in more detail.

The SLICC generation itself was based on an experimental SLICC backend for ProtoGen. We first verified the correctness of this backend and subsequently extended it to support the creation of heterogeneous protocols.

5.5.1 Bridge Generator

At its core, the bridge generator is the result of merging the ideas of HieraGen [11] with HeteroGen [13]. While HeteroGen allows for fusing two protocols to construct a fused directory as described in Section 4.2.1, HieraGen allows for composing a flat protocol into a hierarchical one. The bridge generator brings these two concepts together and joins two different flat protocols into a heterogeneous, hierarchical protocol as described in Section 4.2.2.

5.5.1.1 Input Specification

The generator uses the same input format as ProtoGen. Each file defines a cache coherence protocol in this format by specifying its stable states and the transitions between them for both caches and directories. An abbreviated example of such an input file can be found in Listing 10. It is important to note that each file describes a single protocol. Therefore, a heterogeneous protocol requires multiple files as input, one for each protocol involved.

The format begins by defining the protocol's components. The `Network` struct specifies the virtual networks used by the protocol, equivalent to the `MessageBuffer`

```
Network { Ordered fwd; Ordered resp; Ordered req; };
Cache { State I;  Data cl; } set[NrCaches] cache;
Directory { State I; Data cl; ID owner; } directory;

Message Request {};
Message Resp { Data cl; };
// ...

Architecture cache {
    Stable{I, M}
    Process(I, load, State){
        msg = Request(GetM, ID, directory.ID);
        req.send(msg);

        await{
            when GetM_Ack_D:
                cl=GetM_Ack_D.cl;
                load;
                State = M;
                break;
        }
    }
    // ...
}

Architecture directory {
    Stable{I, M}
    Process(I, GetM, M){
        msg = Resp(GetM_Ack_D, ID, GetM.src, cl);
        resp.send(msg);
        owner = GetM.src;
    }
    // ...
}
```

Listing 10: A cache coherence protocol in the ProtoGen format.

definitions at the top of SLICC state machine declarations. The `Cache` and `Directory` structs define the additional data accessible to their respective state machines. All state machines have two implicit members: `State`, which holds the machine's current state, and `ID`, which stores the component's address in the network. Although `Cache` and `Directory` already have an implicit `State` member,

it seems to be getting redefined in the format. However, this definition does not add a new member but instead sets a default value for the existing `State` member.

Contrary to expectations, the `Message` structs do not directly define message types but instead represent message classes. Their primary role is to specify what additional data messages carry beyond the sender, destination, and message type. This approach mirrors how messages are defined in SLICC. However, unlike SLICC, a separate enumeration of message kinds is unnecessary. Instead, the set of message kinds is inferred from the message names used as the first parameter during message construction.

Following these component declarations, the file includes a block for each controller. These blocks define each controller's complete set of stable states, similar to the `state_declaration` structures in SLICC, except without transient states. In ProtoGen, transient states are generated automatically as needed, eliminating the need to manually construct them. Additionally, these blocks define the transitions each controller can perform in response to incoming messages. These `Process` blocks correspond to the `transition` and `action` blocks in SLICC but do not require separate `in_port` blocks. Unlike SLICC, ProtoGen allows branching control flow within the `Process` blocks. A key difference in ProtoGen is the use of `await` blocks, which allow message handlers to pause and wait for another message. The ProtoGen compiler uses these blocks to determine the necessary transient states, as the `await` operation implicitly maps to a transition into a transient state.

5.5.1.2 Immediate Output

After reading the input files, processing them, and constructing the proxy controller, the generator passes a complete list of controllers to the backend. Each element in this list contains a complete representation of a controller, including a full list of states (encompassing both stable and transient states), all message types that can be received, any additional variables the controller uses, and an extensive list of transitions. The list of transitions is the most noteworthy part of the output, as it encodes most of the controller's logic. Each transition is defined by a start state, an end state, the type of message that triggers it, and a list of

procedural operations to be performed when the transition is taken. These operations are represented as syntax trees originating from the parsed input files and predominantly consist of basic operations such as variable assignments, message constructions, conditional statements, or message send instructions.

5.5.2 Basic Structures

The basic structural elements of SLICC remain largely consistent across different protocols. This uniformity simplifies generating SLICC code, as many foundational constructs can be reused with minimal changes. For instance, the basic structure for state machines presented in Listing 7 remains similar across various controllers, even for different cache coherence protocols. This consistency heavily simplifies the code generation process as only sections encoding protocol logic, such as the `in_port` handlers, require heavy modification between protocols. For this reason, the base layer of the code generator is a templating system that merely fills in a few missing features, such as various names, state identifiers, and protocol-specific parameters. A state declaration, for example, can be generated by taking a template of the overall structure and filling it in by generating one line per state in the list of states provided by the generator. This approach trivially ensures that the generated code conforms to the syntax rules of SLICC while, in essence, not being much more complicated than repeated string concatenation. The same strategy can be applied to most other declarative parts of SLICC, such as data type or message definitions. Unfortunately, the parts of the SLICC file that contain protocol-specific logic like `in_port`, `transition`, or `action` blocks cannot solely be handled by templating and requires a more fine-grained approach.

5.5.3 Event Handlers

Event handling is where the coarse-grained templating approach falls short, requiring a more fine-grained method. The primary input for generating event-handling structures is the list of transitions described in Section 5.5.1.2. In gem5, event handling relies on three primitives: `in_port`, `transition`, and `action`. `in_port` blocks map incoming messages to events, events trigger a `transition` if they

are valid for the current state, and during the transition process every `action` associated with the transition are executed in response.

5.5.3.1 General Structure

While this model seems to align with the generator's input data, there is a critical distinction: in gem5, `action` blocks cannot perform branching control flow. gem5 imposes this restriction due to the way it models resource usage. It will check if enough resources for the transition are available before taking it. This design demands that each transition always performs the same operations. Violating this assumption can lead to performance anomalies, deadlock, or other bugs [25].

This limitation is why SLICC distinguishes between events and messages rather than directly operating messages. Control flow logic is handled by the `in_port` handler, which encodes the results of this logic in the event it emits. For example, many protocols require counting acknowledgments before proceeding with certain transactions. In gem5, this counting logic must reside in the `in_port` handler, which emits different events depending on whether all acknowledgments have been received. In contrast, the input transitions from the generator are not bound by these restrictions and can freely include branching logic. Converting them directly into `action` blocks would require complex syntax tree transformations, pushing all control flow outward to the `in_port` level.

Another challenge is how the individual operations in the operation list should be grouped into actions. In gem5, actions are typically designed to group operations into semantically meaningful and reusable chunks. Unfortunately, the input data from the generator does not provide such semantic grouping. This leaves two options: either generate numerous small, reusable actions that perform individual operations or create one large, single-purpose action for each transition. Both approaches have drawbacks. Generating too many single-operation actions can make the code fragmented and more complex to maintain, while large, single-purpose actions reduce reusability and flexibility.

To avoid the aforementioned issues, the code generator currently processes most of the logic directly within the `in_port` handler. This handler is structured

as a nested if-else chain, where each block checks for combinations of start state and expected message kind. Once a match is found, the corresponding operations for that transition are executed. Instead of relying on gem5's `transition` construct to update the controller's state, we directly modify the current state using the `setState` function. Similarly, messages to other controllers are sent directly from within the `in_port` handler rather than from within an `action`.

5.5.3.2 Allocating Cache Entries

However, certain operations, such as allocating new cache entries or transaction buffer entries, can only be performed from inside of an `action`. This is because these operations depend on implicit state information that gem5 sets while processing a `transition`. To accommodate this restriction, the complete `in_port` handler relies on a few utility actions, as well as matching transitions and events to trigger these actions when necessary.

Triggering an event, however, terminates the current execution of the `in_port`. Therefore, ensuring that these utility actions are only triggered under specific conditions is crucial. They must either be called at the very end of the execution block or invoked before popping the event queue, with a guard in place to prevent them from being triggered again during the following invocation of the `in_port`. This ensures the correct sequencing of operations without prematurely terminating the `in_port` execution or duplicating actions.

5.5.3.3 Dealing with Missing Virtual Channel Information

Another challenge in translating the generator's output to SLICC is that the generator does not specify the specific queues on which any given message may arrive. To address this, the code generator takes a conservative approach, inserting code to handle every possible message in each `in_port` handler. An exception to this rule are the *load* and *store* messages, which can only occur on the `mandatoryQueue`. Since these messages can only be received on this queue, we can safely omit them from the regular `in_port` handlers of other queues. Conversely, as *load* and *store* messages are the only messages originating from the `mandatoryQueue`, we exclude all other message types from being handled there.

5.5.3.4 Dealing with Implicit Memory Operations

Another key difference between the output of the generator and SLICC is how memory operations, such as loads and stores, are handled. The ProtoGen format implicitly assumes that any read or write to a cache line in the directory is a memory load or store. SLICC, however, requires explicit memory load and store messages to be sent over the `requestToMemory` queue. Once these messages are sent, the protocol must stall until confirmation is received on the `responseFromMemory` queue, indicating the success of the operation. To bridge this gap during code generation, we search for read or write operations targeting cache lines in the operation lists when processing the directory's transitions. When such operations are found, they are replaced by the appropriate operation to send a message to the memory queue. The remaining operations are then moved into the `responseFromMemory` handler to ensure that they only execute after the memory operation has been successfully completed.

5.5.4 Generating Bridge Controllers

The key distinction between a bridge controller and a private cache controller lies in their position within the cache hierarchy. A private cache controller typically connects directly to a core, or more precisely, to a sequencer that issues load and store instructions to fetch or update data. These requests arrive via the `mandatoryQueue`. In contrast, a bridge controller interfaces with other caches rather than directly with a core. Consequently, it lacks a sequencer and, by extension, a `mandatoryQueue`. This absence places the bridge in a unique position, as the `in_port` handler of the `mandatoryQueue` typically contains numerous elements crucial for cache functionality: allocating new cache entries, triggering evictions when the cache is full, updating Most Recently Used (MRU) statistics to prevent premature evictions, and more. To maintain the bridge's functionality, these elements must be redistributed to the other `in_port` handlers. Specifically, components related to cache misses (such as allocating a new cache entry and potentially triggering an eviction) need to be relocated to handlers for all transitions that move the bridge from an invalid state (typically the default state) to a non-invalid state. In

protocols of the MSI family, this translates to transitions with *I* as the start state and *GetS* or *GetM* as the incoming message. Conversely, functionality associated with cache hits, like updating MRU statistics, must be integrated into handlers for transitions triggered by the same messages but originating from a valid start state.

5.5.5 Message Flattening

The ProtoGen format allows for the creation of new messages without restrictions. These messages can be sent into a network, accessed like any other struct, or even simply discarded without any side effects. Messages can even be stored in member variables. SLICC, on the other hand, does not allow for the construction of new messages outside of the `enqueue` function. This means that freestanding message construction operations cannot be directly represented in SLICC. To work around this issue, we use "virtual messages". Whenever the SLICC code generator encounters a message constructor, it does not output any SLICC directly but instead adds an entry to an internal data structure, indicating that the current scope contains a virtual message with the respective name and data members. We then add an additional check to the code path responsible for member variable access that determines whether the member belongs to a virtual message. If so, it directly replaces it with the value that was originally given to the skipped constructor. A similar strategy is used to handle message send instructions, with the crucial difference being that the message to be sent must always be a virtual message.

Handling messages that are member variables of other structs is more challenging. We decided to flatten the message into the struct that would have contained it by replacing the message with all its members, prefixing their names with the name of the message itself. One consequence of this strategy is that all assignments to the replaced variable must also be replaced by a series of assignments for all previous members of the message. Unfortunately, this approach can lead to a significant amount of potentially unnecessary code if not all members of the stored message are actually used. To alleviate this issue, we added a filter that

allows us to mark which members of a variable are actually used later, enabling us to discard the rest.

5.5.6 Order-preserving Stalling

The ProtoGen format can be used to specify cache coherence protocols that rely on ordered networks. Such systems operate under the assumption that messages exchanged between controllers follow first-in-first-out (FIFO) semantics. When generating SLICC for such protocols it is important to preserve the correct ordering for messages when they affect the same cache line. While each `MessageBuffer` can easily be configured to maintain message order by setting `ordered=True`, it is equally important that controllers themselves do not inadvertently reorder messages.

One common area where message reordering can occur is during stalling. The most popular strategy for handling stalls in SLICC is *recycling*, where messages that cannot be processed immediately are dequeued and reinserted at the end of the queue after a delay. While this approach increases concurrency in protocols that are resistant to reordering, we cannot assume that the protocols we are working with are reorder-resistant. Therefore, a more conservative approach is needed to ensure correctness without significantly reducing concurrency.

In this scenario, the `stall_and_wait` function provided by gem5 is a more suitable choice. This function allows the controller to stall all messages associated with a specific cache line, preserving the order of messages related to that address. At the same time, it avoids stalling transactions for other cache lines unnecessarily. The main drawback of this method is that messages from stalled addresses must be explicitly woken up once the controller is ready to process them.

The challenge here is that we need to statically determine which state transitions will allow the controller to process previously stalled messages. This problem is further complicated by the fact that stalling is the default response to any message that cannot be appropriately handled in a given state. To address this issue, we take a broad approach: the controller wakes up stalled messages every

time it transitions to a new state. This ensures that any messages waiting to be processed are re-evaluated whenever the controller's state changes.

More in-depth information about the various ways stalling can be implemented in SLICC can be found in Section 5.6.2. Section 6.3 contains an evaluation of the performance impact of choosing to use `stall_and_wait` over `recycle`.

5.5.7 Functional Access Permissions

Another key difference between the ProtoGen format and SLICC is that SLICC requires states to have functional access permissions (see Listing 7), while the ProtoGen format lacks an equivalent concept. As a result, we must generate these permissions during the code generation phase. In general, all transient states must have `Busy` as their access permission, so only stable states must be considered when generating access permissions.

For controllers with attached sequencer (private caches), these access permissions can be trivially derived for any state from its transitions. For example, if a cache state has a self-transition for a *load* event but not for a *store* event, it is assigned `Read_Only` permissions; similarly, self-transitions for both events indicate `Read_Write` permissions. If it has neither the is `Invalid` as its access permission.

However, this heuristic does not apply to directories, as they do not handle CPU-side *load* and *store* events. Additionally, the functional access permissions for directories are often inverted. For example, when a cache line is in the M (Modified) state in a directory, it typically means that the line is outdated in the directory and should not be read or written, as a more recent copy is being modified in one of the caches ($\Longrightarrow$ `Maybe_Stale`). Conversely, when the directory is in the I (Invalid) state, it indicates that the directory is the sole owner of the cache line and can read or write to it without coherence issues ($\Longrightarrow$ `Read_Write`). To correctly determine access permissions for directories, the generator must calculate the union of access permissions from all possible cache states associated with a given directory state. In MSI-family protocols, this often corresponds to the access permissions of the cache state with the same letter designation.

This framework can also support compound states used by fused directories. In these cases, the compound state is broken down into its component states, and the access permissions for each component are calculated according to the protocol it originated from. These permissions are then merged using a union operation to produce the final access permissions for the compound state.

Bridges, which merge both cache and directory functions, require a slightly different approach. Since a bridges combines a cache and a directory, the directory-side permissions must be inverted before merging them with the cache-side permissions. This merge is performed using an intersection rather than a union. The intuitive reason for using an intersection is that the cache and directory components in a bridge operate in layers, creating an "and" relationship. Data must pass through both layers, unlike in merged directories, where components are side by side, allowing data to pass through either layer, leading to an "or" relationship.

5.6 Additional Topics

5.6.1 Functional Access Functions

While gem5 only has the *Atomic* and *Timing* memory access types for general usage, it internally also supports the *Functional* access type. On a base level, functional accesses work like atomic accesses in that they happen instantaneously. However, unlike atomic or timing accesses, they can coexist with both other access types in the same system. Their purpose is to interact with the simulated system from the outside. This is used, among other things, to load binaries into the simulation or inspect and potentially change data inside the simulated system with tools such as a remote debugger. As these accesses originate from outside of the system instead of from inside, all components that could contain the requested or updated part of the memory must be able to handle these requests. In practice, this means that message queues must inspect their content to check for in-flight data blocks, and caches must check if they contain the targeted address and whether or not their copy is valid. The simulator then performs a depth-first

search until it finds all the required places to resolve the request. Implementation-wise, this means that all structures in the cache system, including messages, must correctly implement the `functionalRead` and `functionWrite` functions. Failing to do so generally results in a crash of the simulator due to a failed functional read or write.

5.6.2 Stalling Functions

Most cache coherence protocols must stall while waiting for pending replies. Since stalling essentially results in idle cycles, it can significantly degrade protocol performance. Therefore, minimizing the duration and frequency of these stalls is crucial to maintaining protocol efficiency. gem5 provides several mechanisms to handle stalling, each with its own set of trade-offs. Selecting the appropriate stalling approach is important for balancing performance and complexity. This section explores the advantages and disadvantages of the various stalling options available in SLICC and discusses their impact on cache coherence performance.

5.6.2.1 Vertical Stalling

The simplest option is to have a self-transition for the current state that only calls an empty action. The empty action is necessary as gem5 requires each transition to have at least one action. It is also essential that the empty action is called exactly `z_stall`. Any other name will cause the action to not be marked as protocol stall and cause gem5 to immediately start another attempt at processing outstanding messages, most likely leading directly back to the self-transition. This results in a spin-lock-like behavior and will eventually trigger the deadlock detector if the lock is not resolved fast enough. The spin-lock-like behavior also significantly slows down the simulation and pollutes execution traces. A big problem with this type of stalling is that it blocks the entire `in_port`, preventing even messages targeted at entirely unrelated cache lines from being processed. The advantage of this type of stalling is that it is easy to implement, and its primary weakness, limited concurrency, helps with debugging.

5.6.2.2 Message Recycling

A more advanced method of implementing stalling involves using the `recycle` function of the relevant queue. This function removes the message at the head of the queue and reinserts it at the end after a configurable delay. The primary advantage of this approach is that it allows subsequent messages to bypass the stalled one, thereby reducing the impact of stalling while still keeping the implementation relatively simple. Most of the cache coherence protocols included in gem5 utilize this method for handling stalls. However, there are several drawbacks to this approach. First, it is not particularly realistic. Second, the reinserted message will always trigger another attempt to process it, even if the system state has not changed since the last check, leading to unnecessary retries. Most importantly, this method alters the ordering of the messages in the queue, which can introduce concurrency but may also cause correctness issues if the cache coherence protocol is not designed to handle message reordering.

5.6.2.3 Blocking Addresses

The final option for implementing stalling is to use the `stall_and_wait` function. This method temporarily blocks all messages for a specific address and moves them to a separate queue dedicated to stalled messages. As a result, the stalled messages temporarily "disappear", allowing subsequent messages for other cache lines to be processed. To resume processing of the stalled message, the `wakeUpBuffers` or `wakeUpAllBuffers` functions are used to move messages from the blocked queue back to the front of the main queue. This ensures that the relative ordering between stalled messages and those still in the queue is preserved. Additionally, this method offers the best performance in terms of simulation efficiency. However, the correct placement of the wake-up functions is critical, and misplacing them could lead to messages being indefinitely stalled and eventually causing a deadlock.

5.6.3 Request Routing

Whenever a cache component sends a request to a higher level, it must determine which controller at that level is responsible for the requested cache line. Most protocols achieve this by using the `mapAddressToMachine` function. This

function takes an address and a `MachineType` (e.g., `MachineType:Directory`) as its parameters and returns a `MachineID`, which can then be used to send messages to the corresponding controller. To support this process, each Ruby controller maintains a list of address ranges, which can be configured like other properties in the configuration script. When a controller invokes `mapAddressToMachine`, gem5 iterates through all controllers of the specified `MachineType`, comparing the given address against each controller's list of address ranges. If a match is found, the `MachineID` of that controller is returned. If no match is found, the `MachineID` of the controller with the highest `version` number is returned by default.

This method works well in scenarios where a unique controller is responsible for each cache line. For instance, when routing requests to directories, the request can be directed to the directory associated with the memory controller responsible for the cache line. However, in other scenarios, there may not be a clear correlation between an address and a specific controller. For example, assigning specific memory ranges to shared caches may not make sense, as the goal is for each private cache to send its request to the topologically nearest shared cache rather than assigning them by address range.

To address these cases, gem5 provides the `mapAddressToRange` function. This function builds upon the previous one by also accepting a cluster ID and the number of controllers per cluster as additional parameters. These values are typically set in the configuration script. If multiple controllers of the requested type exist within a cluster, the function distributes the load evenly among them by using the lower bits of the address to compute an intra-cluster ID.

6 Evaluation

In this chapter we are evaluating cache coherence protocols produced by our generator against some of the cache coherence protocols included in gem5.

6.1 Setup

The latest commit of the gem5 version we used to run these experiments is `e8bc4fc`[1]. We built gem5 with support the the X86 instruction set and one of the following Ruby protocols: `MOESI_CMP_directory`, `MESI_Two_Level`, or one of our custom, generated protocols.

For the configuration of gem5 we used the following values unless noted otherwise:

Setting	Value
CPU Model	X86SimpleTiming
Clock Speed	2 GHz
Cores	8
DRAM Type	DDR3 1600 8x8
DRAM Size	8GB
Memory Model	timing
L1 Cache Size	16kB
L2 Cache Size	128kB

For our performance evaluation, we used several benchmarks from the Splash-4 [26] benchmark suite, a modernized version of the popular Splash-2 [27] benchmark suite. gem5, and all benchmarks were compiled with gcc 11.4.0. The exact parameters used for the individual benchmarks can be found in Appendix A.

6.2 Performance of Generated Flat Protocols

We used our generator to generate a flat MESI and MOESI protocol to compare their performance against two handwritten protocols from gem5's standard library: `MOESI_CMP_directory` and `MESI_Two_Level`.

[1] https://github.com/gem5/gem5/tree/e8bc4fc137a5a7e82b601432271a027b652ae69b

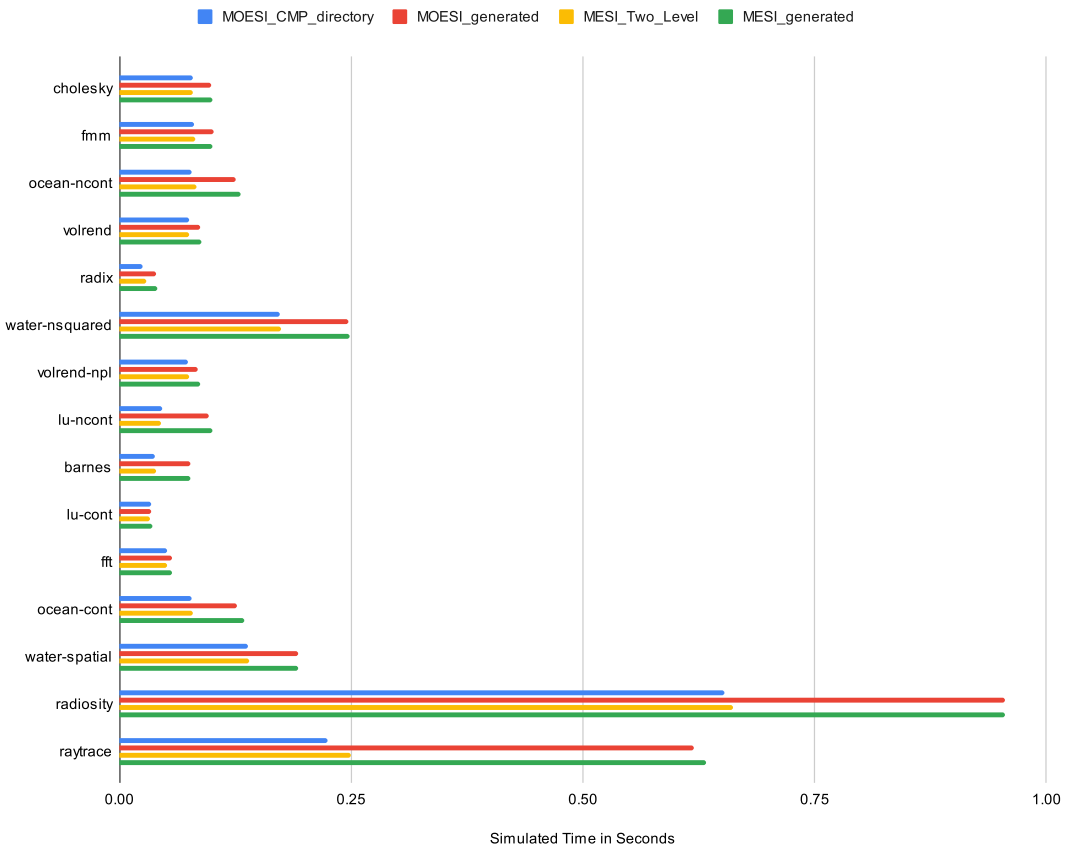


Figure 6: Performance comparison between generated and handwritten protocols.

This evaluation aims to collect a baseline of how our protocols fare against handwritten protocols before introducing heterogeneity. As our generator joins bridges and L2 caches into a single controller, we do not have access to standalone L2 caches for use in this benchmark. Unfortunately, the only protocol included with gem5 that does not use at least one level of shared caches is `MI_example`, which is unsuitable for this comparison as a baseline due to its lack of an S state. As a result, we had to use protocols with L2 caches but tried to minimize the impact of the L2 caches on the test results in the following ways:

- We did not use the clustered setup described in Section 5.4.4. Instead, we used a flat hierarchy where all 8 L1 caches are directly connected to the directory, a point-to-point network with uniform latencies. For the handwritten protocols,

we inserted a *single* L2 cache between the L1 caches and the directory to act as a transparent last-level cache that functionally acts as the directory.
- We limited the size of the L2 cache to 128kB (*l1size* × *cores*). We chose not to reduce the size any further as `MESI_Two_Level` is a strictly inclusive cache coherence protocol, and decreasing the size further could negatively impact the total cache capacity of the L1 layer and cause unnecessary evictions.

Additionally, the handwritten protocols also split the L1 into seperate instruction and data caches. We decided to set both the instruction and data cache to the same 16kB size as the mixed cache we used for our generated protocols, putting them at a slight disadvantage from a total capacity standpoint.

While the results (Figure 6) show that the protocols included in gem5 were almost always faster than our protocols, the relative difference is not uniform across all benchmarks.

We speculate that this performance discrepancy can be explained by how many memory accesses the additional L2 cache prevents. To investigate this hypothesis, we compared the relative difference in memory accesses between the best (`lu-cont`; delta: 7%) and worst (`raytrace`; delta: 254%) performing benchmarks for the two MESI variants.

We observed 3.76 (r: 3.94, w: 2.89) times more memory accesses during the execution of `raytrace` and only 1.39 (r: 1.51, w: 1.20) times more accesses for `lu-cont`. These numbers seem to confirm that L2 efficiency is at least partially responsible for the performance gap. We believe that introducing L2 caches in the form of our bridge controllers would, therefore, significantly improve the competitiveness of our protocols.

6.3 Performance Impact of Preserving Message Ordering

In Section 5.5.6, we described our efforts to implement stalling with a function that preserves the ordering between messages rather than the usual method, which does not. To verify that always preferring the reorder-free stalling implementation is not hurting the performance of protocols that could also utilize the `recycle` function,

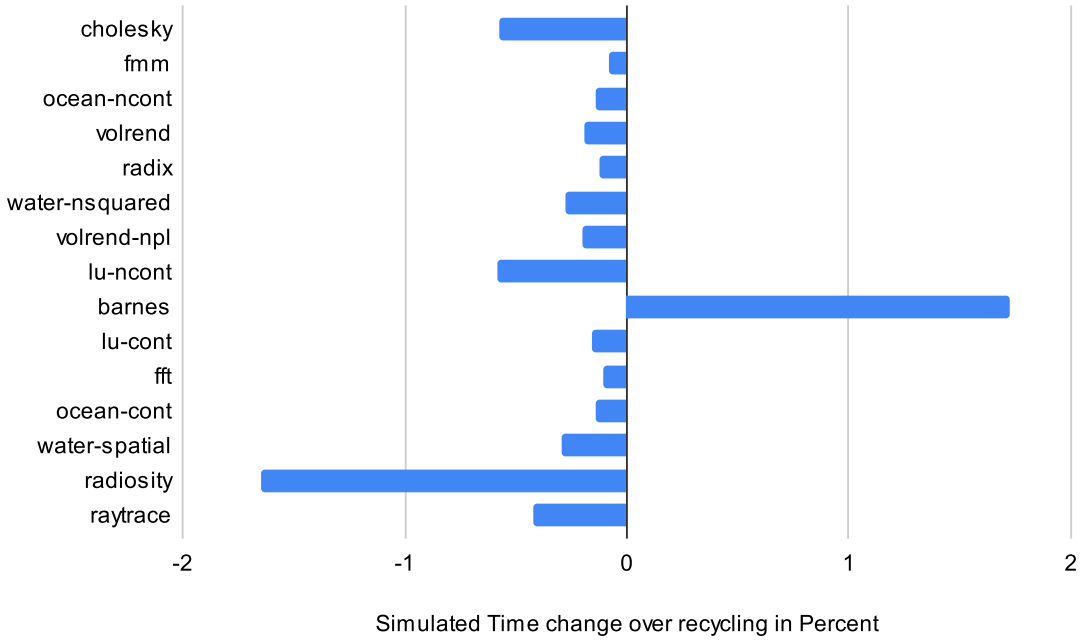


Figure 7: Performance comparison between stalling styles for the same protocol.

we retested the generated MESI protocol using `recycle` instead of the standard `stall_and_wait`. We found that in our tests, `stall_and_wait` is typically between 0%-1% *faster* that `recycle`, validating our choice of stalling implementation. A full comparison of between the two runs can be found in Figure 7.

6.4 Performance of Heterogenous Protocols

Unfortunately, we have not yet been able to benchmark any heterogeneous cache architectures. This section covers the reasons preventing us from testing our heterogeneous protocols and outlines the challenges that must be resolved before these systems can be successfully benchmarked.

6.4.1 Fused Directory

We can not test the performance characteristics of a fused directory-based heterogeneous cache system, as attempting to use more than one kind of private cache simultaneously consistently crashes gem5. When trying to simulate this configuration, one cache always attempts to read a value far outside the normal memory range. Despite our best efforts, we were unable to locate the source of this issue

within a reasonable time frame. As the fused directory architecture is ultimately only a stepping stone toward the superior bridge architecture, we decided not to investigate further.

In a homogenous setup (e.g., using a MESI+MSI directory with only MESI or MSI caches), the performance values measured were almost identical to the values measured with a single protocol directory (*MESI_generated*/*MOESI_generated* in Figure 6).

6.4.2 Bridge

At the time of writing, the protocols generated by our generator still contained deadlocks at the protocol level, preventing us from successfully benchmarking them. However, these protocols do **not** appear to suffer from the same issues we observed with the fused directory variants, as they are able to run past the point where fused directory protocols typically crash.

6.5 Correctness

To ensure the correctness of our generated protocols, we also verified the output of the individual benchmarks whenever possible (in a separate run).

Some of the tests, such as `radix`, offer a command line flag to automatically verify the correctness of the calculation. Other tests, such as `fft` offer merely to print the calculation result to the console. In these cases, we compared this output against the output of running the same test with the same flags outside of gem5 on native hardware.

On top of this verification, gem5 also offers limited deadlock detection that automatically fails a test in case of a deadlock.

7 Summary

In this thesis, we first configured the gem5 simulator to model a NUMA-like system using a single homogeneous cache coherence protocol. This initial setup enabled us to successfully run the entire Splash-4 benchmark suite on our simulated system, establishing a baseline for future experiments. We also implemented basic strategies for controlling data placement within the system without relying on gem5′s Full System mode, laying the groundwork for more targeted benchmarking with tighter control over operations involving shared memory regions.

Next, we extended our simulation to include heterogeneous systems. To generate the necessary heterogeneous cache coherence protocols for these simulations, we developed an SLICC generator. Initially, the generator only supported homogeneous, flat cache coherence protocols, but we expanded its functionality to also support heterogeneous protocols based on fused directories or bridges.

Our key findings indicate that gem5 is likely a suitable tool for exploring the performance characteristics of heterogeneous architectures. First, we confirmed that gem5′s System Emulation mode can handle a variety of standard benchmarks, including those that require direct control over shared memory regions. Second, we demonstrated that multiple Ruby cache coherence protocols can be run within a single gem5 instance, confirming that heterogeneous systems can be simulated without the need for multiple gem5 instances running in parallel.

Additionally, our automatically generated flat protocols not only function correctly but, in some cases, already match the performance of manually written protocols. Lastly, we demonstrated that extending the SLICC backend to support heterogeneous controllers is feasible, addressing one of the final challenges in simulating heterogeneous systems. The integration of bridge controllers, once fully operational, is expected to resolve many of the performance issues we observed in the generated protocols when compared to the baseline.

The source code for the gem5 configuration scripts is available at `https://github.com/sidit77/cxlgen-gem5` and the source code of the SLICC backend is part of `https://github.com/Gingerbreadz/CXLGen`

8 Future Work

8.1 Testing of the Bridge Controller

At the time of writing, the generated bridge controllers still contain bugs on the protocol level that prevent them from being used successfully. Ideally, the SLICC backend is complete and does not require any further changes to generate correct SLICC code for bridge controllers, even after the remaining bugs in the protocol are fixed. However, once that has happened, it is still necessary to thoroughly test the emitted SLICC code to ensure that it is able to successfully run all benchmarks in our suite without triggering deadlocks or other bugs.

8.2 More Benchmarks

Expanding the range of benchmarks used to evaluate our generated protocols is another important area for future work. Currently, our system is evaluated only with benchmarks from the Splash-4 suite. To gain a more comprehensive understanding of the protocols' effectiveness across different scenarios, we plan to incorporate benchmarks from the Parsec [28] and Phoenix [29] suites in future evaluations. Besides these general-purpose benchmarks, we would also like to support more targetted benchmarks that can use the external memory detailed in Section 5.2.2. A benchmark that seems like a good candidate for this task would be a distributed key-value store that keeps its data entirely on the external memory.

8.3 Improved Code Quality of the Generated SLICC Code

Improving the quality of the automatically generated SLICC code is another area for enhancement. Currently, the SLICC backend produces code that contains considerable redundancy in many areas, leading to significantly increased file size compared to equivalent hand-written protocols. This redundancy makes debugging and manual adjustments more challenging, reducing the utility of the generated protocols as a starting point for protocol explorations in directions that our generator does not support.

The most impactful improvement to code quality would be to filter which message handlers are inserted into which `in_port`, thereby limiting the code to handle only the message types that will actually arrive at each `in_port`. Implementing this change could reduce the number of lines of code per controller by up to 66%, significantly reducing dead code in the output.

List of Figures

Figure 1: The basic state machine of a MSI cache 13
Figure 2: A coherence flow for acquiring read permissions. 14
Figure 3: Diagram of this works target system architecture. 26
Figure 4: A comparison between the *Fused Directory* and *Bridge* architectures. 27
Figure 5: Diagram of network architecture in our heterogenous simulation setup. Latency-Ordering: Component (Green) < Local (Blue) < Remote (Red) 47
Figure 6: Performance comparison between generated and handwritten protocols. 63
Figure 7: Performance comparison between stalling styles for the same protocol. 65

List of Listings

Listing 1: Instantiation of System, CPUs, and Memory for a gem5 simulation setup. 31
Listing 2: Instantiation of the Ruby cache system for a gem5 simulation setup. 32
Listing 3: Instantiation of the Garnet network for a gem5 simulation setup. . . 34
Listing 4: Instantiation of the executed binary for gem5 simulation setup. 35
Listing 5: Manually mapping memory into the address space of a simulated program. 38
Listing 6: The message definition for a Ruby protocol. 39
Listing 7: The definition of a Ruby cache controller. 39
Listing 8: The message handling of a Ruby cache controller. 42
Listing 9: The implementation of Ruby directory controller. 44
Listing 10: A cache coherence protocol in the ProtoGen format. 49

Appendix A: Splash-4 Parameterization

Benchmark	Command Line
fft	`./FFT -p8 -m16`
cholesky	`./CHOLESKY -p8 < inputs/tk14.O`
lu-ncont	`./LU-NOCONT -p8 -n256`
lu-cont	`./LU-CONT -p8 -n256`
radix	`./RADIX -p8 -n200000`
barnes	`./BARNES < inputs/n1024-p8`[2]
fmm	`./FMM < inputs/fmm.8.2048`[3]
ocean-ncont	`./OCEAN-NOCONT -p8 -n130`
ocean-cont	`./OCEAN-CONT -p8 -n130`
radiosity	`./RADIOSITY -p8 -ae 5000 -bf 0.1 -en 0.05 -batch`
raytrace	`./RAYTRACE -p8 -m34 inputs/teapot.env`
volrend	`./VOLREND 8 inputs/head-scaleddown4 2`
volrend-npl	`./VOLREND-NPL 8 inputs/head-scaleddown4 2`
water-nsquared	`./WATER-NSQUARED < inputs/n512-p8`
water-spatial	`./WATER-SPATIAL < inputs/n512-p8`

The initial parameters were based on the recommended input as suggested by the *README* included with the source code of Splash-4[4] and were adjusted on demand to keep the runtime of the simulator within reasonable limits ($<$ 30 Minutes).

[2]Copy of `n16384-p8` with reduced n

[3]Copy of `input.8.16384` with reduced n

[4]https://github.com/OdnetninI/Splash-4

Bibliography

[1] T. P. Morgan, "CXL And Gen-Z Iron Out A Coherent Interconnect Strategy." Accessed: Oct. 11, 2024. [Online]. Available: https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/

[2] D. Das Sharma, R. Blankenship, and D. Berger, "An Introduction to the Compute Express Link (CXL) Interconnect," *ACM Comput. Surv.*, vol. 56, no. 11, Jul. 2024, doi: 10.1145/3669900.

[3] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, "Coherence Basics," in *A Primer on Memory Consistency and Cache Coherence*, Cham: Springer International Publishing, 2020, pp. 9–16. doi: 10.1007/978-3-031-01764-3_2.

[4] "Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4," no. 825743. 2024.

[5] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, "Directory Coherence Protocols," in *A Primer on Memory Consistency and Cache Coherence*, Cham: Springer International Publishing, 2020, pp. 151–190. doi: 10.1007/978-3-031-01764-3_8.

[6] "ARM Cortex-A Series Programmer's Guide for ARMv7-A," no. 4. 2014.

[7] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, "Advanced Topics in Coherence," in *A Primer on Memory Consistency and Cache Coherence*, Cham: Springer International Publishing, 2020, pp. 191–209. doi: 10.1007/978-3-031-01764-3_9.

[8] "AMBA CHI Architecture Specification," no. IHI50. 2024.

[9] J. Alsop, M. Sinclair, and S. Adve, "Spandex: A Flexible Interface for Efficient Heterogeneous Coherence," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 261–274. doi: 10.1109/ISCA.2018.00031.

[10] N. Oswald, V. Nagarajan, and D. J. Sorin, "Protogen: automatically generating directory cache coherence protocols from atomic specifications," in *Proceedings of the 45th Annual International Symposium on Computer*

Architecture, in ISCA '18. Los Angeles, California: IEEE Press, 2018, pp. 247–260. doi: 10.1109/ISCA.2018.00030.

[11] N. Oswald, V. Nagarajan, and D. J. Sorin, "HieraGen: Automated Generation of Concurrent, Hierarchical Cache Coherence Protocols," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 888–899. doi: 10.1109/ISCA45697.2020.00077.

[12] M. Wang, T. Ta, L. Cheng, and C. Batten, "Efficiently Supporting Dynamic Task Parallelism on Heterogeneous Cache-Coherent Systems," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 173–186. doi: 10.1109/ISCA45697.2020.00025.

[13] N. Oswald, V. Nagarajan, D. J. Sorin, V. Gavrielatos, T. Olausson, and R. Carr, "HeteroGen: Automatic Synthesis of Heterogeneous Cache Coherence Protocols," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 756–771. doi: 10.1109/HPCA53966.2022.00061.

[14] J. Lowe-Power *et al.*, "The gem5 Simulator: Version 20.0+," *CoRR*, 2020, [Online]. Available: https://arxiv.org/abs/2007.03152

[15] A. Gutierrez *et al.*, "Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 608–619. doi: 10.1109/HPCA.2018.00058.

[16] A. Hansson, N. Agarwal, A. Kolli, T. F. Wenisch, and A. N. Udipi, "Simulating DRAM controllers for future system architecture exploration," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, IEEE Computer Society, 2014, pp. 201–210. doi: 10.1109/ISPASS.2014.6844484.

[17] M. M. K. Martin *et al.*, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, Nov. 2005, doi: 10.1145/1105734.1105747.

[18] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, 2006, doi: 10.1109/MM.2006.82.

[19] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 33–42. doi: 10.1109/ISPASS.2009.4919636.

[20] S. Bharadwaj, J. Yin, B. Beckmann, and T. Krishna, "Kite: A Family of Heterogeneous Interposer Topologies Enabled via Accurate Interconnect Modeling," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6. doi: 10.1109/DAC18072.2020.9218539.

[21] C. Lameter, "NUMA (Non-Uniform Memory Access): An Overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors.," *Queue*, vol. 11, no. 7, pp. 40–51, 2013.

[22] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble Page Management for Tiered Memory Systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, in ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 331–345. doi: 10.1145/3297858.3304024.

[23] S. R. Dulloor *et al.*, "Data tiering in heterogeneous memory systems," in *Proceedings of the Eleventh European Conference on Computer Systems*, in EuroSys '16. London, United Kingdom: Association for Computing Machinery, 2016. doi: 10.1145/2901318.2901344.

[24] P. Balcer, "Exploring the Software Ecosystem for Compute Express Link (CXL) Memory." Accessed: Sep. 06, 2024. [Online]. Available: https://pmem.io/blog/2023/05/exploring-the-software-ecosystem-for-compute-express-link-cxl-memory/

[25] J. Lowe-Power, "Modeling Cache Coherence with Ruby: In port code blocks." Accessed: Oct. 02, 2024. [Online]. Available: https://www.gem5.org/documentation/learning_gem5/part3/cache-in-ports/

[26] E. J. Gómez-Hernández, J. M. Cebrian, S. Kaxiras, and A. Ros, "Splash-4: A Modern Benchmark Suite with Lock-Free Constructs," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*, , Ed., Austin, TX (USA): IEEE Computer Society, Nov. 2022, pp. 51–64. doi: 10.1109/IISWC55918.2022.00015.

[27] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, in ISCA '95. S. Margherita Ligure, Italy: Association for Computing Machinery, 1995, pp. 24–36. doi: 10.1145/223982.223990.

[28] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[29] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 13–24. doi: 10.1109/HPCA.2007.346181.