Generating Gem5 Cache Coherence Controllers with ProtoGen

Theo Olausson

MInf Project (Part 2) Report

Master of Informatics
School of Informatics
University of Edinburgh

2021

Abstract

This paper presents an extension of the *ProtoGen* software for cache coherence protocol synthesis. Motivated by the insight that performance evaluation is a crucial aspect of developing new cache coherence protocols, the presented extension gives ProtoGen the ability to generate cache controllers suitable for use with the cycle-accurate system simulator *Gem5*. Thus, the proposed software extends ProtoGen's reach from protocol synthesis and verification to quantitative evaluation of protocols. To showcase the usefulness of this tool, the paper presents two case studies, the results of which challenge conventional wisdom in cache coherence protocol design and thus highlight the tool's ability to quickly establish results which are specific to a given system.

Acknowledgements

Thank you Vijay, for guiding me in my work even as the circumstances made progress challenging. Thank you CAPS for the weekly Wednesday morning coffee break, which was often a much-needed slice of normality during these times of working from ”home”, as well as for all your help on my project. Finally, thank you to my family and friends for helping me stay sane this past year.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

That developing cache coherence protocols is a challenging, laborious, and error-prone process is a fact familiar to any computer architect who has dabbled in the memory sub-system. Different data sharing patterns, cache hierarchies, and races on the interconnections between cores cause all sorts of headaches when it comes to making sure that the protocol will act correctly in every possible scenario, making us as designers reliant on formal verification techniques such as model checking. To make matters worse, correctness is not the only point of concern. Once it has been determined that the protocol will not blow up the computer or, even worse, lead to extremely subtle bugs that only show up once in a blue moon, we still have performance to worry about. Obtaining data about the coherence protocol's performance will require us to once again implement it, this time in a simulator instead of a formal verification tool. Not only does this add to the already laborious process, but we also need to be confident that this new implementation is in fact equivalent to the one that was verified; if not, we are back to square one as far as safety is concerned.

Over the past few years, the world of computing has increasingly gone in the direction of specialization. Domain-specific accelerators are driving progress in genomics, image processing, and several other fields. Just as quickly as the capabilities of the algorithms driving progress within these fields evolve, so do their expectations of the hardware they run on; now, perhaps more than ever, the biggest challenge faced by the computer architect is that of the clock, and we need tools which can relieve us of the most laborious and time-consuming tasks. Better tooling for memory consistency and cache coherence can help architects design systems which incorporate these accelerators in the shared-memory domain, enabling rapid progress further up the stack.

1.2 Project Goals & Overview

The overarching goal of this two-part Master of Informatics project is to facilitate the design, construction, and verification of cache coherence protocols, as part of the

ongoing ProtoGen project (Oswald et al., 2018, 2020).

1.2.1 Synopsis of Part 1

In the first part of the project (Olausson, 2020), we investigated the *stable state protocols* which the ProtoGen software takes as input. The project began by defining a concrete correctness criterion for stable state protocols, so that their correctness could be verified automatically. Then, this correctness criterion was used to derive a taxonomy of the ways in which stable state protocols may be incorrect: message bugs, which ”relate to the specification of which messages to send at what point in the protocol” (Olausson 2020, p.25); state bugs, which ”relate to the setting and updating of state in the protocol” (Olausson 2020, p.25); high-level design bugs, which ”boil down to a lack of fully understanding the target system and how to design a protocol for it” (Olausson 2020, p.25); and concurrency bugs, which form the ”complement of the union of the above three classes of bugs” (Olausson 2020, p.26). Using insights from this taxonomy, two algorithms were then presented for bug localization: Reduced Model Checking, which finds message bugs in protocols at the granularity of a single transaction, and State Consistency Tracking, which finds state bugs by throwing an error if an inconsistent state update is found. In experimental evaluation, these two algorithms were found to be capable of finding bugs at a significantly finer granularity than standard model checking, which opened up for future work on synthesising bug fixes. Finally, several future directions of work were suggested, all of which had in common the fact that they would further the tool’s ambition to accurately locate, and ultimately synthesize fixes to, the bugs present in the stable state protocols.

1.2.2 Contributions of Part 2

For the second part of the project, we build a new backend for ProtoGen so that the synthesised protocols can, in addition to being verified in a model checker, also be evaluated holistically with the Gem5 simulator. Concretely, the contributions of this second part of the project are as follows:

- An extension of the ProtoGen codebase, which translates ProtoGen’s internal representation of the synthesised protocols into cache controllers suitable to be used with Gem5 (chapter 3)
- An extensive evaluation of this new backend (chapter 4), which includes:
  - A sanity check that the generated protocols are capable of supporting fine-grain simulation (section 4.1)
  - A first case study, which compares the performance of three commonly used protocols (section 4.2)
  - A second case study, which investigates the impact of handshake messages and stalling in systems with unordered interconnects between the cores (section 4.3)

It is noteworthy that this second part of the project does not follow the list of future work which was identified at the end of the previous dissertation. Instead, we view the contributions of this second part as being in the same space as those of the first, but in an orthogonal direction. The motivation for this was that the authors felt further work along the same direction as the first part of the project would be hard to fairly evaluate without engaging in large-scale user studies. Such user studies would give a much better view of, for example, whether an algorithm which can synthesise simple bugs in stable state protocols actually helps architects be more productive. However, ProtoGen as a tool is very much still in its adolescence, and does not yet have the breadth of users needed to carry out such studies. This is precisely the motivation for this shift in focus: by integrating ProtoGen with a popular system simulator, the tool may see additional uptake in both industry and academia, thus enabling well-justified and fairly evaluated future research in this field.

1.3 Outline of the Report

We begin this short report by discussing cache coherence protocols, ProtoGen, Gem5, and prior work in this space in chapter 2. We then discuss the proposed backend in chapter 3, highlighting both conceptual as well as real issues with translating ProtoGen's internal representation of protocols to Gem5's SLICC language. The previously discussed evaluation of the backend is then presented in chapter 4. Finally, chapter 5 summarizes what has been presented in this work, discusses the most time-consuming challenges faced and how they might have been avoided, and outlines future work.

Chapter 2

Background

Before diving into the contributions of this dissertation, we will first spend some time familiarizing the reader with the underpinning concepts. Section 2.1 introduces *cache coherence protocols*, first discussing their raison d'être as a solution to problems with programming *shared memory architectures*, and then going deeper into their design and implementation. Next, section 2.2 discusses *ProtoGen*, the coherence protocol synthesis software which this project builds upon, and section 2.3 discusses the *Gem5* simulator as well as its domain-specific language for cache coherence controllers, *SLICC*. Finally, section 2.4 discusses prior work in this space.

> **Disclaimer.** By necessity, several sections in this chapter overlap with the corresponding sections in the report from the first part of the project (Olausson, 2020). We invite the curious reader to also read the background chapter there, since some topics are discussed in more (or less) detail than here.

2.1 Shared Memory Architectures and Cache Coherence Protocols

A *shared memory architecture* (Valduriez, 2009) is a system consisting of several processing elements, all sharing the entire main memory. This has become a common feature in modern computer systems, which typically consist of several processors working in parallel, either on independent workloads or together to achieve a joint task. In the latter case, the processors of course need a way to synchronize their actions and communicate their results; shared memory allows them to do exactly this by simply reading and writing to agreed addresses.

Shared memory architectures come with many benefits: they are both intuitive to program and avoid the cost of moving data between each processor's private regions of memory. The downside, however, is that most of these benefits are only true if you look at them from 30,000 feet. In reality, out-of-order execution within each processor and non-zero latencies on the interconnects between them make programming shared-memory architectures a bug-prone headache, and data kept in a processor's private

caches must one way or another be propagated to everyone else in order for the value to be communicated.

The first of these problems is typically hidden from high-level programmers through the use of a *memory consistency model* (Nagarajan et al., 2020), which is effectively a contract between the hardware and the software regarding which interleavings of memory operations may occur; as long as the programmer writes their parallel program using the synchronization primitives offered to them by their chosen high-level language and the compiler abides by this contract, the headaches will be avoided. Memory models are a complex and fascinating subject on their own, but are not the focus of this dissertation, and we will not discuss them further.[1]

To resolve the latter issue, shared memory architectures make use of *cache coherence protocols*, which are precisely the topic of this dissertation. Informally, a cache coherence protocol dictates what happens when a load, store, or eviction occurs in a private cache, so that the value is propagated to others (when needed) and in a way such that the memory consistency model is not violated.[2] Of course, determining when and how to propagate data values to other caches will have an impact on the system's performance, and thus performance is an important metric to consider when designing a cache coherence protocol for a new system. Furthermore, there are many more dimensions to the design space than one may first expect.

2.1.1 The Design Space of Cache Coherence Protocols

Cache coherence protocols are typically described as extended finite state transducers. Each state is associated with a permission to the cache line, for example whether the core may read or write to the line; transitions correspond to internal (evictions, loads or stores) or external (incoming messages) events, and may also include guards on auxiliary variables; taking a transition may not only change the state, but also output messages to other machines or update auxiliary variables such as counters. Thus, one important way in which cache coherence protocols may differ is in the sets of states and transitions they define; as will see in the next section, the set of states defined by a protocol is in fact often used as the protocol's name.

Another way in which cache coherence protocols may differ is whether they are *snooping* or *directory-based*. In snooping protocols, the caches and the memory controller are connected together by a single bus; messages sent between machines are simply broadcasted over this bus. This makes the system design rather simple, but may not scale well to large numbers of caches due to the ensuing bandwidth requirements of the bus. Directory-based protocols, on the other hand, scale better by using point-to-point networks between the machines. In order to keep track of which messages should

[1] The topic of memory consistency models is certainly more complicated than our swift tour would suggest; for example, high-level languages themselves often have formal memory models, which would then be a contract between the programmer and the compiler! For more information, see Nagarajan et al. (2020).

[2] This informal definition of a cache coherence protocol should be sufficient for this Part 2 of the project; for a slightly more formal take, see the corresponding chapters in Olausson (2020) and Nagarajan et al. (2020).

be delivered to which machine, a special *directory* controller is used; this keeps track of information such as which caches currently hold a certain cache line with read or write permission, so that when a request comes along, it can be forwarded to the right machines.

Finally, directory-based cache coherence protocols can differ in whether the point-to-point networks between the machines are ordered or not; that is, whether messages may arrive at their destination in a different order than the one in which they were sent. This is an important distinction, because unordered networks may be easier to construct in hardware, but lead to more races on the interconnects. Thus, protocols using unordered networks typically need to make use of additional messages – so called *handshakes* – to avoid succumbing to these races; we will look at this in more detail in section 4.3.

We have now seen three dimensions along which cache coherence protocol designs may differ. In this dissertation, however, we will only focus on the first and the last of these, because the ProtoGen tool which forms the basis of our work only works for directory-based protocols. Let us now look at the first dimension – the different states and transitions defined by the protocol – in more detail, as it is a rather important topic for the reader to be familiar with.

2.1.2 Protocols Used in This Dissertation: MI, MSI and MESI

In this dissertation, we will often refer to three different protocols: MI, MSI, and MESI (also called the Illinois protocol (Papamarcos and Patel, 1984)). In this section, we will spend some time familiarising the reader with each of these protocols as well as highlighting the differences between them, so that they understand why one might perform better than another on a given workload.

MI is the most simple cache coherence protocol one could devise, consisting of only two stable states: Modified (`M`) and Invalid (`I`).[3] The `I` state simply does what it says on the tin: it indicates that the given address is not present with read nor write permission in the current cache. The `M` state, on the other hand, indicates that the cache line is present, and that the line may be read or written to. The `M` state is exclusive, meaning there may only be one cache holding a given line in `M` state. Thus, if a cache currently holds a line in `I` state but wants to read or write to it, it must first obtain the line in `M` state. The cache may then hold the line in `M` state until it is evicted due to a cache line conflict, or another cache wants to read or write to the line. Figure 2.2 summarizes the transitions between the states in the cache and directory controllers.

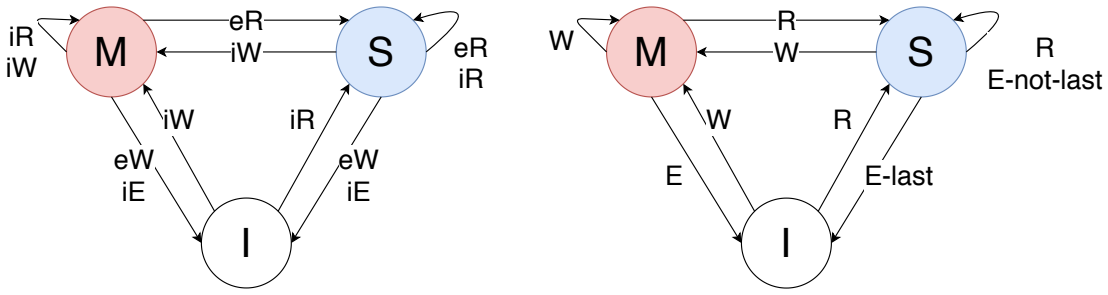
(a) MI cache controller (b) MI directory controller

Figure 2.1: MI protocol. Legend: R = Read; W = Write; E = Eviction; i = internal (originating in the machine itself); e = external (originating in some other machine).

[3] The Modified state could just as well be called the Valid state, but we will refer to it as the Modified state to make the link to MSI and MESI clearer.

But what about read-only data? Imagine a scenario in which a set of processors are executing a multi-threaded program, and all threads share a variable storing some configuration parameter of the workload. Once its initial value has been set up, no thread ever writes to the variable again, but they frequently read it. In MI, such sharing of read-only data between processors is extremely expensive; since there is no distinction between read and write permissions, each cache would have to obtain an exclusive copy of the data every time it was to be read, leading to the variable being bounced around each cache.

The need for better handling of read-only data is the motivation for the MSI protocol, which adds the Shared (`S`) state. In this state, the cache line is present, but may only be read – not written. If a cache is holding a line in `S` state but decides it wants to write to it, it must first upgrade the line to `M` state. Since it is guaranteed that the value will not change in the cache while it is held in `S`, there may thus be arbitrarily many caches holding a given line in `S` at any one time; however, if there is a cache which holds the line in `M` state, then there must not be any sharers. This is shown in detail in Figure 2.2.



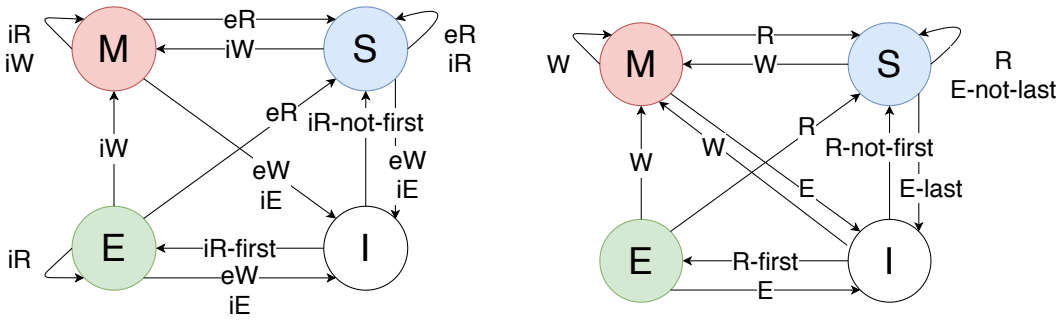
(a) MSI cache controller

(b) MSI directory controller

Figure 2.2: MSI protocol. Legend: R = Read; W = Write; E = Eviction; i = internal (originating in the machine itself); e = external (originating in some other machine). Note that the directory needs to differentiate between when the last sharer evicts the line and when there are still other sharers holding the line. Diagram and caption obtained from Olausson (2020), with permission.

Finally, the MESI protocol adds the Exclusive (`E`) state, which a cache enters only when it requests a line with read permission and there are no other readers of the line at this current time (Figure 2.3). Like the `M` state, only one cache may hold the line in this state at a time; but unlike the `M` state, it only gives the processor the permission to read the line. What is then the point of this state? Is it not just a degenerate version of the `S` state? The difference is that when a cache holds a line in the `E` state, it knows that no one else is. Thus, if it decides it needs to upgrade to `M` state to obtain write permission, it can do so silently – without sending any messages to anyone else at all! If such scenarios are common, e.g. if data is largely private and often accessed in a read-then-write manner, then avoiding these extra messages for the upgrade in permission may save significant time.

Thus far, we have described these protocols on a very high level, discussing how the different *stable* states are connected but not how to actually implement these transitions. For example, we have said that when an MI-style protocol is used and a cache



(a) MESI cache controller

(b) MESI directory controller

Figure 2.3: MESI protocol. Legend: R = Read; W = Write; E = Eviction; i = internal (originating in the machine itself); e = external (originating in some other machine). Diagram and caption obtained from Olausson (2020), with permission.

wants to read a line it holds in `I`, it must first obtain the line in `M`. But what exactly does this mean in practice? In a directory-based protocol, such a transition would typically require the cache to send a certain message – in this case often called a `GetM` – to the directory. The directory might then either fetch the data from RAM, or forward the request to another cache if there already is someone else who holds the line in `M` (a so called ”owner”). Then, the directory or the owner will reply to the requesting cache with a message containing the data. It is only upon receiving this last reply that the requester may proceed with updating its state to `M`, so while it is waiting it should use some intermediate state – say `IM`. Such an intermediate state, or *transient* state, is often excluded from high-level discussions of the protocols, but will in reality be a vital part of the protocol.

2.1.3 A Brief Note on Implementing Cache Coherence Protocols: Transaction Buffer Entries

Cache coherence protocols are typically implemented in hardware, although software-based approaches are possible as well. The exact way in which this is done is not a topic we will cover in this dissertation, but it is important to note that transistor area is always a key constraint when designing a real piece of hardware. As one might expect, implementing the full extended finite state machines described above for every cache line in every cache would therefore be prohibitively expensive; in fact, even storing the state variable could become expensive if there are a lot of possible states!

To minimize the cost of implementing the coherence protocols, we can instead make use of (*Transaction Buffer Entries*) (TBEs). These are special cache entries which are reserved for the use of transactions which are currently in flight, and hold the auxiliary counters and transient states. All that is necessary to store in the cache lines themselves is then the stable states, of which there are typically very few (2-4, meaning we require 1-2 bits of extra space per cache line). Since only a small subset of the cache lines are likely to have use for the extra variables at any one time, we can often get away with using a number of TBEs which is significantly lower than the number of cache lines, hence saving a lot critical space.

2.2 ProtoGen

ProtoGen (Oswald et al., 2018, 2020) is a software which automates away a lot of the heavy lifting involved in writing cache coherence protocols. Given an atomic specification of a cache coherence protocol, a very simple version of the protocol where transitions induced by races on the interconnect networks are not present, ProtoGen can generate a fully concurrent protocol. Thus, ProtoGen can be a vital part of any computer architect's toolbox as it can massively speed up the development cycle for cache coherence protocols.

It is easiest to understand how ProtoGen derives the transitions needed to handle races on the networks by looking at an example. Suppose you have a cache controller implementing an MSI protocol, and that the cache wants to evict a line it currently holds in `M` state. It sends a special message called `PutM` to the directory, and should now enter a transient state `MI` whilst it waits for the directory to acknowledge the replacement. Now suppose that whilst this is happening, a different cache wants to read the same line. It sends a `GetS` (the meaning of which should be obvious from the naming convention) to the directory, which may arrive before the first cache's `PutM` has. The directory sees the `GetS` and sends a `Fwd_GetS` to the first cache as it is the current owner of the line. When the cache in `MI` receives this `Fwd_GetS`, what should it do? At this point, the directory has decided that the transaction started by the second cache attempting to read the line should, logically, happen before the replacement. The cache needs to respect this ordering; hence, the cache should act *as if it received the* `Fwd_GetS` *before evicting the line.* Therefore, we can deduce its behaviour from what it would have done had the `Fwd_GetS` arrived in `M`, which is typically to demote itself to the `S` state before replying with the latest value of the cache line to the requester; the only exception is that at the end of the day, we still want to model the fact that the eviction has happened, so instead of going to `S` we go to `SI`, as if we had first gone to `S` but then immediately evicted the line. Continuing with this type of reasoning, we can derive the correct action for every possible race on the networks.

Having constructed an internal representation of the full protocol, ProtoGen can output code in a model checker called Murϕ (Dill, 1996). This allows the protocol to be formally verified for correctness; however, Murϕ is not a system simulator, and as such does not allow for any evaluation of the generated protocol's performance. Adding such capabilities to ProtoGen is precisely the topic of this dissertation.

2.3 Gem5

Gem5 (Lowe-Power et al., 2020) is an open-source computer systems simulator which can be used to evaluate the performance of new architectural features. Originally a merger of the GEMS and m5 simulators, Gem5 has over the years been extended with several additional features such as GPU modelling and DRAM power usage analysis, and several leading companies in the industry now contribute to its further development.

Gem5 supports two simulation modes: *Syscall Emulation* (SE) and *Full System* (FS).

SE mode is the simpler of the two, focusing on modelling the CPU and memory subsystem and hence not requiring the user to configure the rest of the hardware. However, SE mode therefore only runs user-level code; even worse, it offers very limited support for user-mode parallelism (e.g. pthreads). In FS mode, Gem5 instead emulates the entire hardware system. This allows Gem5 to execute unmodified OS kernels and gather more extensive data on the system performance, but slows down the process and requires more work from the user to set up the system.

2.3.1 Modelling cache coherence protocols with SLICC and Ruby

SLICC (*Specification Language for Implementing Cache Coherence*) is Gem5's domain-specific language for writing cache coherence protocols. The protocols are compiled into Ruby[4], which is Gem5's detailed simulation model of memory, caches, interconnection networks and coherence controllers.

In SLICC, each unit in the memory system requires a controller to be defined for it in its own separate *.sm* file. In SE mode, this typically means that, at minimum, each level of cache (e.g. L1 and L2) will have a controller defined for it. Additionally, if the protocol is directory-based then a controller for the directory will be required, too. For FS mode simulation, one also needs to define a controller for IO and Direct Memory Access (DMA). These latter controllers are typically non-caching and quite simple themselves, but introduce additional transitions in the directory and cache controllers; as such, full system simulation protocols are often larger and more complicated than those found in textbooks.

SLICC mandates an event-driven programming paradigm, where receiving messages or CPU requests trigger events which in turn cause transitions depending on the current state of the controller. As a result, a SLICC controller can be split into four largely independent parts, as shown in Figure 2.4. This is so that the Gem5 simulator can make certain guarantees about the controller's behaviour, for example that transitions are always executed atomically (thus requiring Gem5 first to check that all of the resources required to complete the transition's actions are available). As we will see later, this leads to some challenges when automatically generating the controllers from ProtoGen's internal representation.

Finally, Ruby requires the user to write a configuration script for each protocol. This script is quite simple in nature: its purpose is to show Ruby how the different controllers are connected, to designate the networks between them and whether they are ordered or not, to set hyperparameters such as the sizes and access latencies of the caches, et cetera.

2.4 Related work

To the author's knowledge, there have been no previous attempts at generating Gem5/SLICC controllers from a simple DSL. However, the wider space of cache coherence protocol

[4]Not to be confused with the programming language carrying the same name.

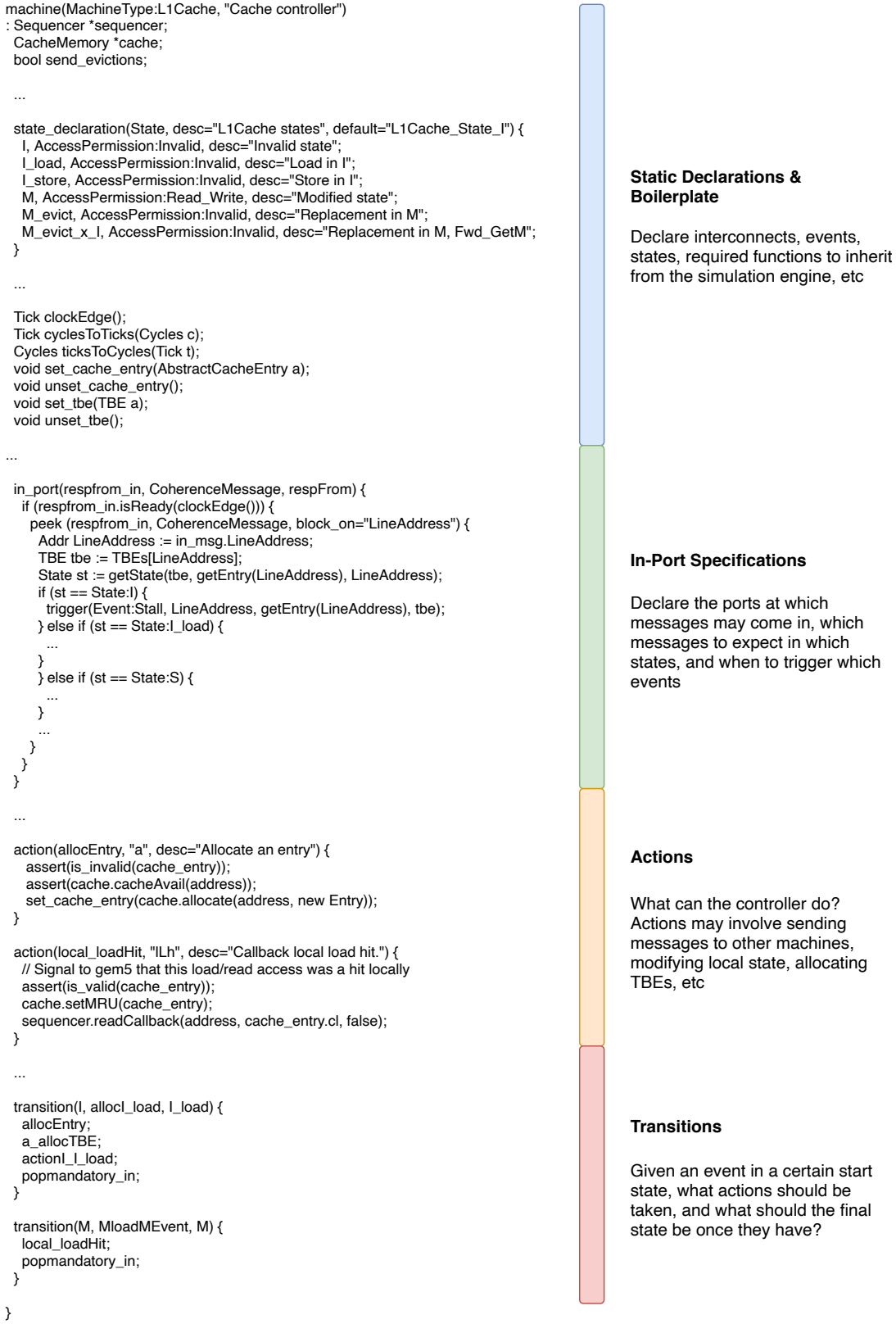


Figure 2.4: The structure of a general controller in SLICC. The relative sizes of the sections in this diagram do not reflect what a full protocol specification might look like.

design and verification has seen quite some interest in the literature, including in recent years. We will now discuss some of these efforts.

Very recently, the Hemiola framework for cache coherence protocol development was proposed by Choi (2021). Hemiola is embedded in the Coq interactive theorem prover (Barras et al., 1997), offering an alternative to the traditional method of verifying cache coherence protocols with model checking. By interleaving the protocol design process with the proof of the protocol's correctness, verification becomes a first-class citizen in the development phase. Once designed and, by construction, hence also proven correct, a protocol written in Hemiola can be compiled to the hardware synthesis language Kami (Choi et al., 2017). Thus, Hemiola covers the entire process of designing a cache coherence protocol, from its initial inception to the construction of the hardware. Certainly, one can imagine compiling Hemiola protocols to SLICC, as well. We consider Hemiola to be a significant step forward in this space, although time will have to tell whether its extensive reach and rigorous process will be offset by the labour cost stereotypically associated with interactive theorem proving.

Earlier work in the literature included some attempts at synthesising protocols from partial specifications, most notably VerC3 (Elver et al., 2018) and TRANSIT (Udupa et al., 2013). A common feature of these efforts is that they, unlike ProtoGen, approach the problem as a typical program synthesis task, rather than focusing exclusively on the introduction of concurrency into the system (as ProtoGen does). This results in the state space of possible complete protocols blowing up; VerC3 attempts to deal with this by heuristically pruning the search space, while TRANSIT instead requests more information from the user in the form of solutions to counter-examples found by the synthesis engine. Either way, the cost of this increased generality is more labour from the user – either because they must specify larger portions of the protocols, or because they must address the counter-examples.

Going even further back, there were some attempts at making the design process less laborious, even if not using synthesis. Teapot (Chandra et al., 1996) was a domain-specific language in which the user could construct cache coherence controllers, which would then be compiled to a general-purpose language (C) in a form suitable for model-checking. Although no synthesis was involved, Teapot did lessen the overhead involved in programming the controllers.

Chapter 3

The SLICC Backend

In this chapter, we cover the meat of the matter: the SLICC backend for ProtoGen. We begin in section 3.1 by discussing the task on a conceptual level, outlining how the backend fits into the wider scope of ProtoGen and what challenges are involved in translating ProtoGen's internal representation of a protocol to SLICC. Then, section 3.2 discusses the backend as it is implemented in our codebase, touching upon a few unexpected issues that we faced during the process. Finally, section 3.3 discusses what challenges, issues and limitations remain.

3.1 Conceptual Overview & Challenges

On a high level, ProtoGen is a transpiler from a DSL to a target language; what makes it special is the automated race detection and transient state generation it does in-between. Thanks to its modular structure, adding a new backend, for example to target SLICC, is therefore conceptually an easy task: just add a translation from ProtoGen's internal representation of the protocols to the syntax of the target language, and let the user choose which backend should be invoked.

Figure 3.1 shows how the backend fits into the larger scope of ProtoGen in slightly more detail. First, the user's specification of the protocol, written under atomic assumptions in ProtoGen's DSL, is parsed. This parser turns each transaction described in the protocol into an (extended) finite state machine (recall the discussion in section 2.1.1). Then, different serialization orders of potentially racing transactions are considered, and the state machines are updated with transient states such that no matter how the messages arrive at the controller, the serialized order is respected. Then, finally, these e-FSMs are translated into SLICC. Simple, right? Unfortunately, as is often the case, the devil is in the details.

3.1.1 Managing Main Memory Accesses

In the ProtoGen DSL, the directory is assumed to have instant access to any cache line. That is, if it receives a request for a line, it can immediately reply with the data – as

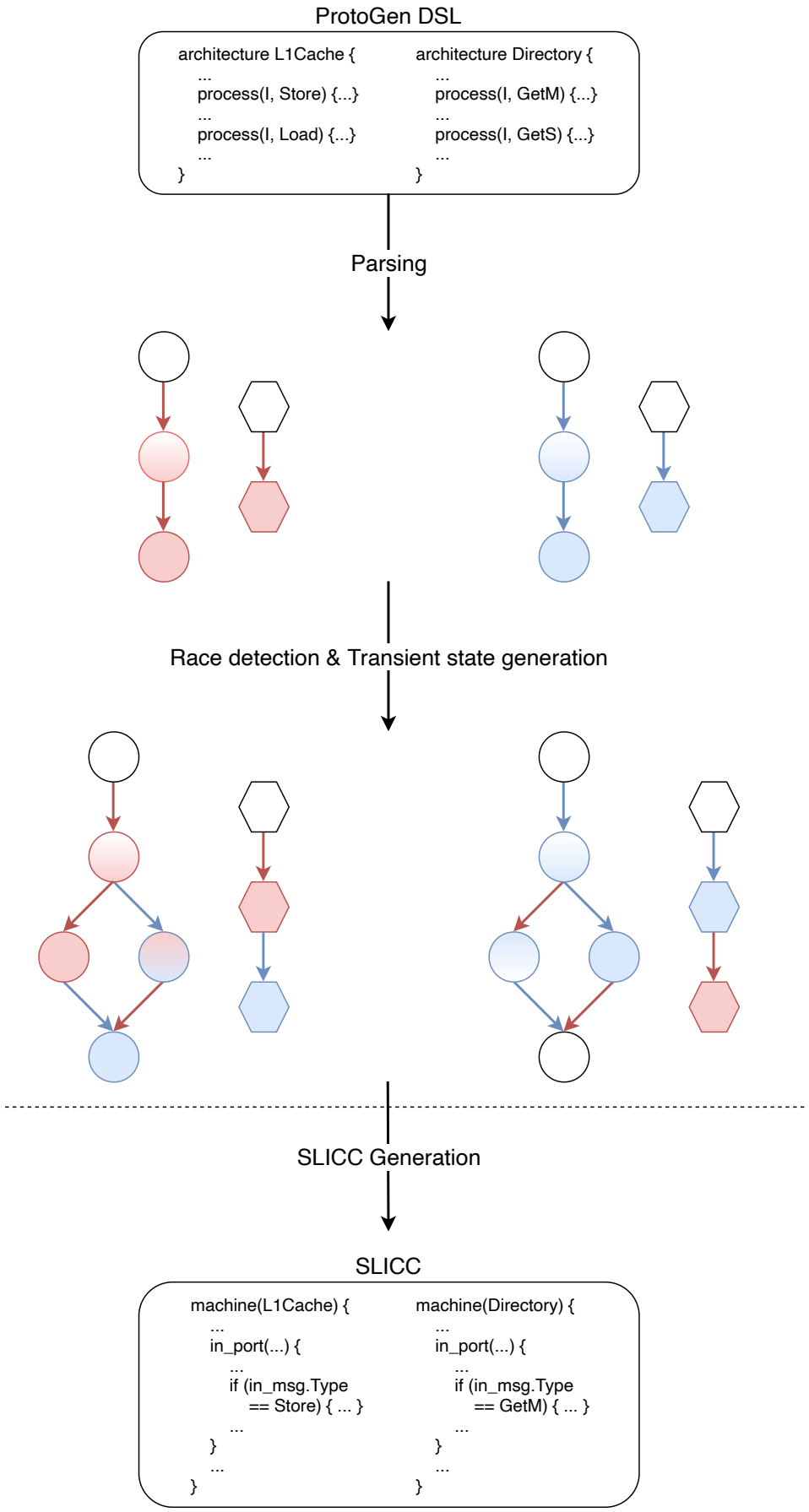


Figure 3.1: Overview of ProtoGen. The frontend parses the input DSL, creating FSMs for individual transactions (cache FSMs with circular nodes, directory FSMs with hexagonal nodes). Then, possible serializations of concurrent transactions are considered, and the FSMs are updated with new transient states to handle the ensuing races. Finally, these FSMs are translated to a target language, in this case SLICC. Everything above the dashed line is prior work.

if it could manifest the data out of thin air. In reality, the directory does not have such magical powers – it must instead ask the main memory for the data before replying.

The way memory reads and writes are handled in SLICC is that the directory is connected to two special ports, `requestToMemory` and `responseFromMemory`. To obtain a cache line, a message of type `MEMORY_READ` is sent on the request network, and some cycles later a message of the same type will appear on the response network, now with the message's `DataBlk` field set to the value of the line. Similarly, writes are handled by sending a `MEMORY_WB`, which will set the value in memory to that of the request's `DataBlk`.

When generating SLICC controllers from ProtoGen's internal representation of a protocol, we must thus make some minor adjustments to the directory's finite state machine. Whenever a directory in state S upon receiving a request message M needs to fetch/write data from/to memory to fulfill the request, we send a request to main memory. Since we must now wait for this request to finish, we move to a new state S_M. Only once the memory has responded do we execute the behaviour that the directory was supposed to execute in the original (S,M) transition, and move to the state S' which the user specified that the (S,M) transition should end in. But what about branches? The (S,M) transition defined by the user may branch on some predicate, and not all branches may require contacting main memory. For example, a common specification of the (`M`, `PutM`) directory transition in an MSI-style protocol is that the data value of the `PutM` should only be written to main memory if the machine which sent the `PutM` is the current owner of the line – that is, the `PutM` is not stale. If we were to disregard this in our implementation as discussed above, then we would sometimes write stale values to memory, which would make programming the computer incredibly challenging. We therefore make a small adjustment to the process above: instead of simply going to the S_M state and later executing all of the expected logic there, we scan the logic of the (S,M) transition and find, for each read and write, the *closest ancestral branch*. That is, we find the point at which the program most recently branched before this read or write. Then, for each closest ancestral branch of a memory read or write, we let the directory execute the transition up to this point *before* sending the memory request. To keep track of which branch to resume execution at once the memory controller replies, we simply give each ancestral branch its own, unique transient state S_M_i (for some unique natural number i).

3.1.2 Dealing with Direct Memory Access

In section section 2.3.1, we mentioned that Full System (FS) simulation in Gem5 requires us to set up Direct Memory Access (DMA) controllers. Without this DMA, the system has no way of, for example, reading files on disk. Unfortunately, the ProtoGen DSL does not model DMA. Although the DMA controller itself is rather simple and could more or less be statically generated without the user's input, the presence of DMA events would also force us to change the directory's FSM. We could leave it out all together and focus on Syscall Emulation (SE) mode simulation, but this would severely limit the usefulness of the backend due to the SE mode's inability to run arbitrary concurrent workloads.

To overcome this obstacle, we make a simple adjustment. Instead of generating a typical write-through, non-caching DMA controller, we generate a DMA controller which looks identical to the cache controllers. This way we do not rely on the user to specify any more information, and do not complicate the directory's FSM since, from the point of view of the directory, the DMA is simply another cache. It is noteworthy that this "hack" does have an impact on the system evaluation – such caching DMA controllers are not typically used in practice, and depending on the data access patterns of the DMA, attaching a cache to it may either benefit or be detrimental to performance in our simulations. However, we argue that this is not a cause of concern; architectural performance evaluation typically focuses exclusively on CPU-bound pre-defined Regions-of-Interest (ROI) in the target programs, which will not invoke IO and therefore will not be affected by the performance of the DMA controller.

3.1.3 Particularities of the Programming Paradigms

The biggest issue in translating ProtoGen's internal represention of a protocol into SLICC is that the ProtoGen DSL is very liberal in terms of how the protocol should be specified, allowing for arbitrary interleaving of branches, local state changes, and messaging. This is well suited for the original task of generating Murphi code, because Murphi is similarly relaxed. As we briefly touched upon in section 2.3.1, SLICC, on the other hand, mandates an event-driven programming paradigm. Recall that SLICC protocols can be divided into four parts: the mostly boilerplate and static declarations; the in-ports, where we define which messages may arrive where, and invoke corresponding events; the action blocks, which implement conceptual tasks such as sending a particular type of message to the current cache line owner; and the transitions, which dictate which actions should be executed given the current state and the type of the event which was triggered. Furthermore, although the official Gem5 documentation makes no mention of this, it turns out that all branching must be handled in the in_ports:

> *There are a number of poorly documented "rules" to writing SLICC protocols. One rule is that you should not do *anything* other than trigger a transition (call trigger(<Event >)) in a in_port. [...] Another rule that isn't documented is that the only place you should use a 'if' statement is in in_ports. You should never use an 'if' (to take two different actions) in an 'action' block. [...] If you bypass these mechanisms, strange things happen :).*
>
> – Lowe-Power (2020).

What does this mean in practice? Consider a cache transitioning from `S` to `M` in an MSI protocol. Since the `M` state is exclusive, the cache must collect invalidation-acknowledgements (acknowledging that the sharer has given up read permission to the line) from all other sharers before it can finish its transaction. A natural way of implementing this is to track how many such acknowledgements are expected (supplied to the requester by the directory), along with how many have already been received. When a new acknowledgement arrives, the latter counter is then incremented before being compared to the former. If they match, the cache can safely transition to `M`, knowing that no core has read permission to the line anymore; if they don't, we sim-

ply wait for the next acknowledgement. This way of implementing the transaction is supported by both the ProtoGen DSL and by Murϕ, but in SLICC it would lead to a ”chicken-and-egg”-style problem: per the above rules the increment must be put in an action block, but the following branch must be placed in the in-port; yet it is the in-port specification which will trigger the transition that executes the action block. In order to resolve this paradox, we would have to move the branch above the increment, and change the branch condition to reflect this. Such changes are easy to make manually but auto-generating them in a way that will work regardless of how the protocol is written is challenging, particularly when there are several nested branches or the branch condition is more complex than a simple (in-)equality check.

How can we overcome this mismatch? One way could be to constrain the syntax of ProtoGen's DSL, so that only protocols matching the branches-first flow of SLICC could be expressed in the DSL. However, this is a rather unsatisfying solution; our vision of ProtoGen is to make it a general, modular tool, where backends can easily be added for more target languages (e.g. hardware description languages such as RTL and Verilog). Tailoring the DSL to match SLICC specifically does not sit well with this vision. Furthermore, we would ideally like the output of the SLICC backend to be as similar to that of the Murphi backend as possible, to make it easier to verify their equivalence. Thus we are left with no option but to break the rules, and implement most of the protocol's logic directly in the in-ports. To minimize the risk of ”strange things” happening, we make a few exceptions:

(A) Cache transactions which start in an invalid state, i.e. without the cache line present, trigger an event which first allocates the cache entry.

(B) Cache transactions which end in an invalid state must end by triggering an event which deallocates the cache entry.

(C) Transactions which may require the use of a TBE first trigger an event to allocate one.

(D) Cache transactions which start in a transient state and and end in a stable one (indicating that we have finished making use of a TBE) must end by triggering an event which deallocates the current TBE.

(E) All branches in the in-ports must end with an event being triggered; if none of the above hold, this event is simply a no-op.

(A)-(D) are necessary because allocating and deallocating cache lines or TBEs requires obtaining the resource from the simulation kernel; this is likely to be a common source of resource contention, so to avoid strange behaviour from the kernel we make sure that it has the chance to stall the triggered event. (E) is there to make sure that the kernel makes progress, and that the number of transitions which can be executed within a single cycle is modelled correctly.

3.1.4 Matching Messages to Networks

Finally, one small conceptual challenge we face in this task is that in the ProtoGen DSL, there is no sense of which network a given message belongs to. For example,

there is no explicit semantic link between a network called `requests` and messages which we, as protocol designers, would think of as being requests. In SLICC, this is done a bit differently, because the expected messages to be received on a certain network are defined within that particular network's in-port. This means that, in SLICC, one can for example throw an error if a `Put`-style message is ever received on the request network. You could even use the networks to differentiate the meaning of messages, taking different actions depending on which network a message arrived on despite being in the same state and receiving the same type of message. This is not possible with ProtoGen, because when we in the ProtoGen DSL define a transition, the unique identified of the transition is the state *S* in which it begins and the message or attempted access *M* which triggers it; we do not specify at all which network *M* is arriving on.

To get around this issue, we simply generate the complete list of transitions for every network's in-port. Functionally, this is OK; certain transitions will simply never be invoked when executing the protocol, such as those corresponding to response messages on the request network. However, it does mean that we inherit the ProtoGen DSL's limitation in this regard, and makes the generated controllers appear a bit bloated.

3.2 Implementation

The SLICC backend consists of about 3,300 lines of Python 3.6 code, spread over 58 files. Furthermore, the backend makes use of about 550 lines of template .sm files; these contain skeleton SLICC code, with holes that are dynamically filed in by the backend. This structure is modelled after the Murphi backend for ProtoGen (Oswald et al., 2018), and three of the files (GenSLICCTree.py, GenPCCToSLICC.py, and TemplateClass.py; see Figure 3.1) were adapted from code written by Nicolai Oswald.

The full structure of the backend is shown in Figure 3.1. At the top level, the RunSLICCModular class defines the entry point into the backend. This then invokes ModularCache, ModularDir, ModularDMA and ModularMsg in /ModularSLICC. These in turn use the classes in their corresponding sub-directories (/cache, /dir, /dma, and /msg) in order to generate the target .sm files. For example, the ModularCache will use the GenCacheInPorts class to generate the in-port declaration section of the cache controller.

Unsurprisingly, a lot of the logic involved in for example translating a machine's FSM to an in-port can be re-used across the caches, directory and DMA. As a result, these machine-specific files will in turn often invoke methods in the generic classes provided in the /common subdirectory, such as for example /common/GenMachineInPorts.py. Roughly 40% of the backend's logic is housed in the /common subdirectory[1], which not only explains the surprisingly low line count but also illustrates how important efficient code reuse is for the maintainability of this backend.

[1] Counted by lines of code.

```
$ tree SLICC
SLICC
|-- ModularSLICC
|   |-- ModularCache.py
|   |-- ModularDMA.py
|   |-- ModularDir.py
|   |-- ModularMsg.py
|   |-- SLICCTokens.py
|   |-- TemplateClass.py
|   |-- cache
|   |   |-- GenCacheActions.py
|   |   |-- GenCacheBase.py
|   |   |-- GenCacheEvents.py
|   |   |-- GenCacheFooter.py
|   |   |-- GenCacheHeader.py
|   |   |-- GenCacheInPorts.py
|   |   |-- GenCacheOutPorts.py
|   |   |-- GenCacheStateDecl.py
|   |   |-- GenCacheStaticFns.py
|   |   |-- GenCacheStruct.py
|   |   |-- GenCacheTBE.py
|   |   |-- GenCacheTransitions.py
|   |-- common
|   |   |-- GenMachineActions.py
|   |   |-- GenMachineEvents.py
|   |   |-- GenMachineHeader.py
|   |   |-- GenMachineInPorts.py
|   |   |-- GenMachineOutPorts.py
|   |   |-- GenMachineStateDecl.py
|   |   |-- GenMachineStruct.py
|   |   |-- GenMachineTransitions.py
|   |   |-- GenPCCToSLICC.py
|   |   |-- GenSLICCTree.py
|   |   |-- Util.py
|   |-- dir
|   |   |-- GenDirActions.py
|   |   |-- GenDirBase.py
|   |   |-- GenDirEvents.py
|   |   |-- GenDirFooter.py
|   |   |-- GenDirHeader.py
|   |   |-- GenDirInPorts.py
|   |   |-- GenDirOutPorts.py
|   |   |-- GenDirStateDecl.py
|   |   |-- GenDirStaticFns.py
|   |   |-- GenDirStruct.py
|   |   |-- GenDirTBE.py
|   |   |-- GenDirTransitions.py
```

```
|   |-- dma
|   |   |-- GenDMAActions.py
|   |   |-- GenDMABase.py
|   |   |-- GenDMAEvents.py
|   |   |-- GenDMAFooter.py
|   |   |-- GenDMAHeader.py
|   |   |-- GenDMAInPorts.py
|   |   |-- GenDMAOutPorts.py
|   |   |-- GenDMAStateDecl.py
|   |   |-- GenDMAStaticFns.py
|   |   |-- GenDMAStruct.py
|   |   |-- GenDMATBE.py
|   |   |-- GenDMATransitions.py
|   |-- msg
|       |-- GenDMADefault.py
|       |-- GenMsgBase.py
|       |-- GenMsgEnum.py
|       |-- GenMsgStruct.py
|-- RunSLICCModular.py
|-- TempSLICC
    |-- * various template files *

9 directories, 117 files
```

Figure 3.1: The layout of the SLICC backend code. The contents of the TempSLICC directory have been removed, for brevity.

3.2.1 Unexpected Issues

Even with suitable planning in advance to tackle the conceptual challenges described in the preceding section, we find that we run into several unexpected issues while implementing the backend. Here we detail a few of them.

3.2.1.1 Silently failing CPU callbacks

Early on in the development of the backend, we found that even small-scale SE-mode experiments would consistently deadlock. Inspecting the execution trace, there was nothing that seemed wrong from the point of view of the coherence protocol: messages were being exchanged correctly, state variables were being updated and there was no deadlock to be found on the interconnects.

After a serious amount of digging, it became clear that the problem was that while the coherence interface happily marked off each transaction as it finished, sometimes the CPU would not be made aware that a transaction it had initiated had been completed. Thus, the CPU would wait indefinitely for the coherence interface to get back to it with a cache line it had wanted to access, resulting in a deadlock. The root cause of this was that the callbacks to the CPU – special functions in SLICC which let the CPU know that a load or store access can now go ahead – were placed in the in-ports. This, it turns out, caused them to occasionally silently fail, despite seemingly not requiring any special resources.

Ultimately, this issue was resolved by adding a sixth exception to our in-ports-only approach:

(F) Finishing a cache transaction which started with a load or store coming into the memory system from the CPU always triggers an event, in which the CPU callback must take place.

3.2.1.2 Improperly handled Read-Modify-Write instructions

When simple SE-mode tests started consistently passed, it was time to move on to FS-mode tests. However, attempting to just boot a Linux kernel would consistently fail for multi-processor setups. Although no deadlock or assertion violation was encountered, the simulation would at a certain point of the boot process simply livelock, seemingly continuing to run without ever making any progress. Ultimately, we found that the root cause of the issue was that our in-port definitions were missing a seemingly optional ”block_on” statement. The presence of this statement was not mandated by the compiler, nor was it ever used in the unofficial ”Learning Gem5” e-book (Lowe-Power (2017)); to the author’s knowledge, it is in fact not mentioned anywhere in the Gem5 documentation at all. However, digging into the Gem5 source code, we find that the value of this statement is used when handling atomic Read-Modify-Write (RMW) instructions in the CPU. Since such RMW instructions are an integral part of any synchronization or communication between cores, this would explain why the issue only became apparent when multiple cores were present. Adding a block_on statement to each in-port definition and setting it appropriately, the issue went away and Linux kernels could successfully be booted even in multi-processor setups.

3.3 Discussion: Limitations of the Implemented Backend

Although we consider the completion of this backend a great feat given the great differences between SLICC and the ProtoGen DSL, there are two limitations that are worth highlighting.

The first limitation is that of the configuration files. Recall from section 2.3.1 that SLICC requires the user to define a config for each protocol, which shows Gem5 how to set up the cache system – for example, which networks are ordered and which are not, which machines are connected and how, how big the caches are, and so on. In its current form, the backend does not create such a configuration file for the generated protocol, due to limitations of the ProtoGen DSL. While the ProtoGen DSL does allow the user to specify some of this file's contents – such as the number of caches in the system and the ordering of the networks – most of the configuration lies outside of this limited scope, since it is information that is not needed for the original task of formally verifying the correctness of the protocol. We do not consider this to have a large impact on the usefulness of the tool, however, as these config files are typically very small (100-200 lines of code).

The second limitation is that the extra transient states we generate in order to handle main memory reads (section 3.1.1) must be *stalling*. That is, if the directory receives any other request to the same line while it is waiting to hear back from main memory, that request must be stalled. This limitation is due to the fact that since the memory is assumed to be instantly accessible in the ProtoGen DSL, the user will specify the transition atomically. Thus, we have no information about what to do when another request arrives during this one, since such a scenario would not be possible if the transition was really atomic. This leaves us with no option but to stall for the duration of the main memory request, which can be expensive. In real protocols, such stalls are often mitigated by coalescing requests. For example, if the directory is fetching the data due to a cache wanting to read the line and then receives the same type of request from another cache, it does not really need to wait until the first fetch finishes and then send another `MEMORY_READ`; instead, it could simply note down in some internal structure that the data – once obtained – should be sent not only to the original requester, but also to the new one.

Chapter 4

Evaluation

In the previous chapter, we saw that – with some effort – we can translate ProtoGen's internal representation of protocols into SLICC code. We also saw that achieving this feat required quite a few tricks. In this chapter, we ask the critical question: *do the generated protocols really work?*

We begin in section 4.1 with a very basic test of correctness, where we simply check whether we can boot a relatively modern Linux kernel in full-system simulation. Recalling that the motivation behind this project was to enable quick evaluation of protocols, we then move on to a set of case studies. In the first case study (section 4.2), we compare the performance of the MI, MSI and MESI protocols discussed in section 2.1.2. In the second case study, (section 4.3), we instead fix the overarching protocol to be MSI, but vary some fine-grain details of its implementation to evaluate their impact on its performance. Through these case studies, we hope to illustrate the backend's usefulness.

4.1 Validation: Booting Linux in Full System Mode

Before we proceed with our case studies, we first ought to make sure that the generated cache coherence controllers are capable of full-system simulation at all. The simplest way of checking this is to boot a Linux kernel in a multi-core system with the coherence protocol active; since booting a Linux kernel is a significant workload in itself, a successful boot would indicate that the protocol is capable of running FS mode tests. We could then proceed with our case studies, knowing at least that we do not expect any immediate issues such as incorrectly specified transitions or issues with accessing main memory.

To carry out this validation step, we generate SLICC controllers for the MI, MSI, and MESI protocols (section 2.1.2) and compile them with Gem5 version 20.1 (Lowe-Power et al., 2020). We obtain a Linux kernel of version 5.4.49 from the Gem5 resources repository (gem5.org, n.d.a), and set up a simple system model with the TimingSimpleCPU CPU model; this combination of CPU model and Linux kernel is known to be functioning in Gem5 v20.1 (gem5.org, n.d.b). With the system model

set up, we then start FS simulations for each of the MI, MSI and MESI protocols and check whether the kernel is booted successfully. We perform this experiment with 1, 4, and 16 CPUs present, and repeat each run 5 times, making for a total of $3 \cdot 3 \cdot 5 = 45$ runs.

We find that the Linux kernel is successfully booted in all 45 of the runs. Thus, we conclude that the cache controllers which the backend generate are capable of full system simulation, and may now move on to our first case study.

4.2 Case Study 1: Comparing the Performance of MI, MSI, and MESI

4.2.1 Problem Statement

Recall from section 2.1.2 the motivations of the MSI and MESI extensions of the MI protocol: MSI adds the read-only `S` to facilitate data sharing, and MESI also adds the `E` state to speed up scenarios where private data is upgraded from read-only to write permission. In this case study, we seek to answer the question of whether these optimizations actually improve the performance of the system; and, if so, by how much?

4.2.2 Experiments

To answer this question, we first need to choose a set of benchmark programs to compare the protocols on. In the literature, the benchmark suites PARSEC (Bienia, 2011) and SPLASH-2 (Woo et al., 1995) are popular choices for system evaluation. However, several programs in these two benchmark suites contain little communication between processors and are therefore not well suited to experiments comparing different coherence protocols. We therefore restrict our experiments to the ”cholesky” and ”lu” (contiguous blocks version) programs from SPLASH-2, along with ”bodytrack”, ”freqmine”, and ”swaptions” from PARSEC; these programs have all been identified as containing significant data sharing and/or synchronization in the characterization papers (Bienia, 2011; Woo et al., 1995). Furthermore, we use the versions of these programs provided by Samani et al. (n.d.), where each program has been annotated with Gem5 handlers for its region-of-interest (ROI); this allows us to gather performance characteristics without polluting our results with irrelevant data such as how long it took to boot the system.

Having chosen the set of benchmark programs to evaluate the performance on, the next step is to construct the experimental system. Aiming to make the system as realistic as possible, we use 4 processor nodes connected together in a 2x2 mesh. Co-located with each node is a full L1 and a shard of the directory; each shard of the directory is responsible for a subset of the memory’s address space. This means that a core’s private L1 will always sit in the same node as the core, but requests that need to go via the directory may need to be routed via some other node depending on the address of the request. As we do not generate coherence controllers for L2 or L3 caches, we add

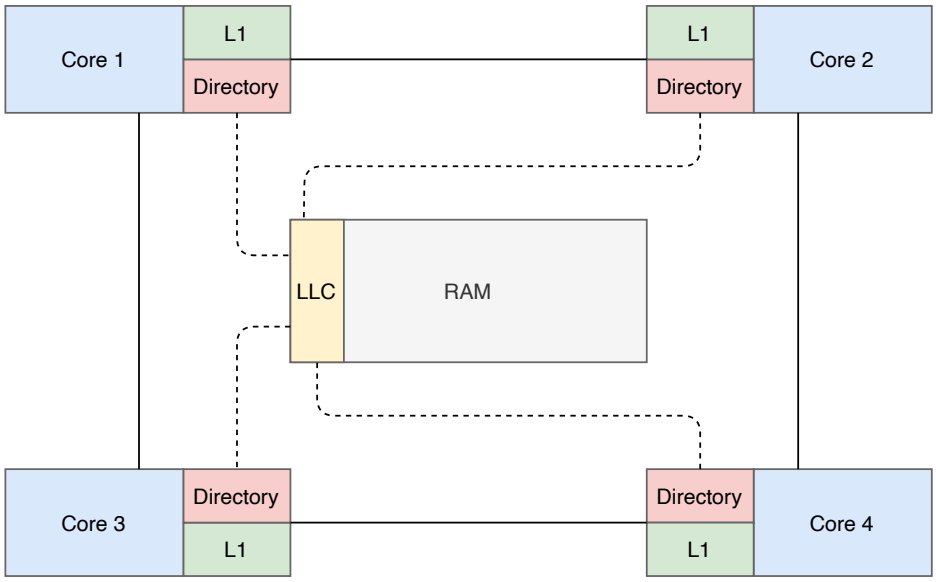


Figure 4.1: The system model used in section 4.2 and section 4.3. Each node contains a CPU core, a private L1 cache, and a shard of the directory responsible for one quarter of the memory range. Nodes are placed in a mesh and connected only to their immediate neighbors (solid lines). Each directory is connected to the LLC (dotted lines), which is co-located with main memory.

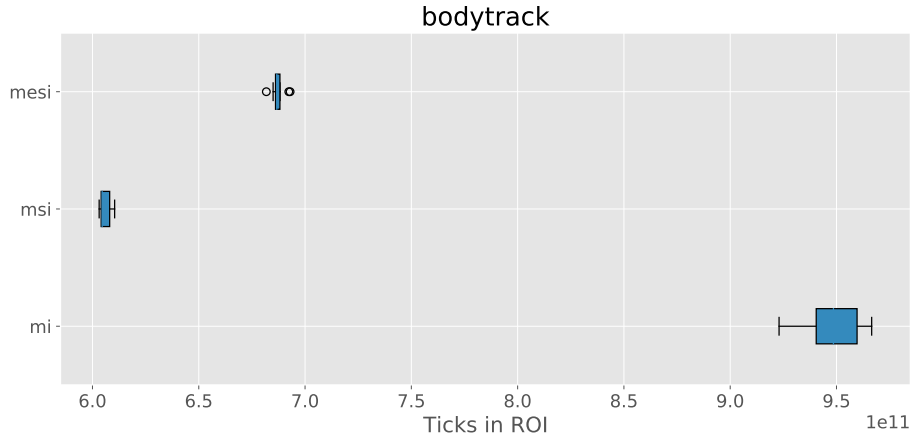
a last-level cache below the point of coherence, between the directory and the main memory; this way we avoid accessing main memory for each L1 miss, which would make our results quite unrealistic. This system model is summarized in Figure 4.1. The hyper-parameters of the caches, such as their sizes and access latencies, can be found in Appendix A.

Having decided on the experimental set up, we may now proceed with evaluating the protocols. We begin by generating the MI, MSI and MESI controllers with ordered networks[1]. For each combination of protocol and benchmark, we boot a full system simulation and note down how many CPU ticks it took to execute the workload's ROI; we do this 10 times for each protocol and benchmark, making for a total of $3 \cdot 5 \cdot 10 = 150$ runs. To avoid waiting for potentially livelocked experiments, we impose a maximum wall clock time of 6 hours for each run.

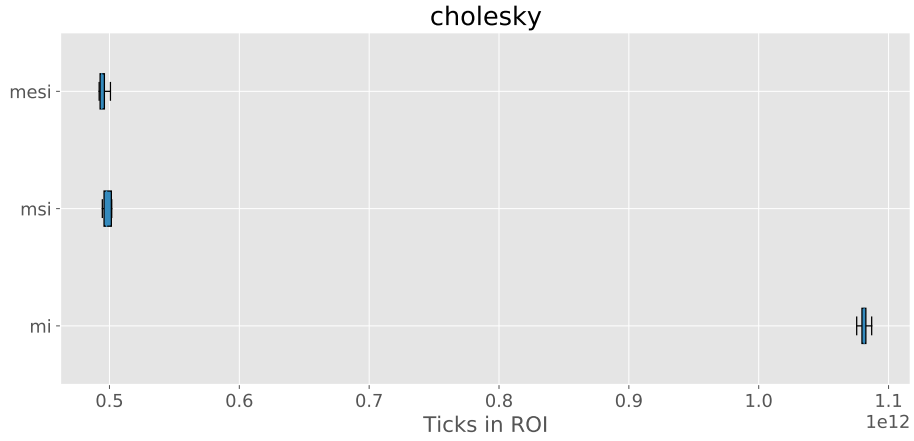
4.2.3 Results & Discussion

We find that 19 out of 150 runs do not finish within 6 hours, which indicates that the exceptions (A)-(F) from chapter 3 do not fully avoid the presence of strange behaviour from the simulation kernel. Figure 4.2 shows the runtimes for the 131 workloads which did finish. As expected, the jump from MI to MSI is very large in all workloads; dropping the `S` state results in an average slowdown ranging from 1.5× in bodytrack to 3.4× in lu_cb. This indicates that the presence of a read-only, non-exclusive state

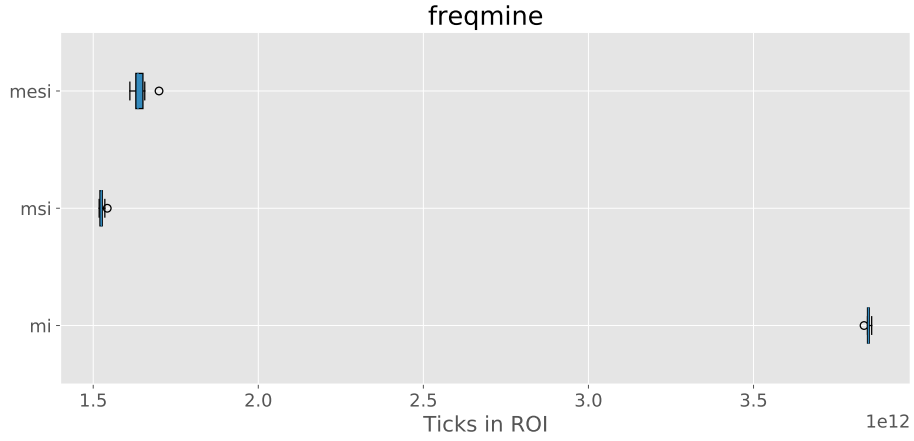
[1]Using fully ordered networks will reduce the variance in our results and allows us to focus on the optimization introduced by each extra state



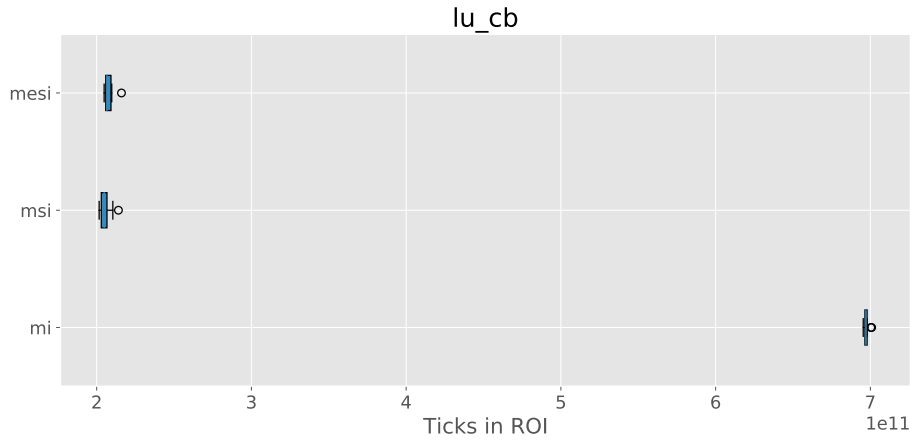
(a) Run-times in ticks for the bodytrack workload.



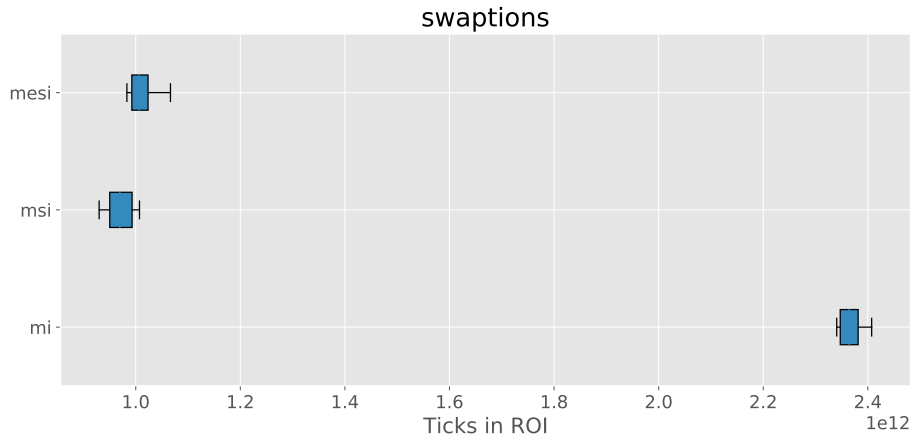
(b) Run-times in ticks for the cholesky workload.



(c) Run-times in ticks for the freqmine workload.



(d) Run-times in ticks for the lu_cb workload.



(e) Run-times in ticks for the swaptions workload.

Figure 4.2: Runtimes within each workload's ROI per protocol considered in section 4.2.

can massively speed up parallel workloads such as those considered here. However, in these workloads we do not find any evidence that the E state leads to better run-times in general; only in cholesky does MESI obtain the lowest average run-time, but even then its distrubtion overlaps significantly with that of MSI. In lu_cb both MSI and MESI perform about equal, and in the rest of the workloads the E state actually appears to slow down the workload considerably. This is especially clear in bodytrack (Figure 4.2a), in which the average run-time is 13% higher for MESI than it is for MSI.

How can this be? On a surface level, one may expect MESI to be at least as efficient as MSI given that it contains all of MSI's states. However, recall that the point of the E state is to speed up private read-write data patterns by letting a cache silently upgrade to write permission when it was the first (and only) cache to obtain read permission. This is only safe to do as long as the cache remains the only sharer of the line; as soon as someone else wants to read the line, we must first demote from E to S, which involves sending a message to the first cache. What we see in this case study is thus that this added communication is prevalent in several of these workloads, and that it in fact outweighs the benefits of the silent permission upgrade.

4.2.4 Conclusion

In conclusion, there are three findings in this case study. The first is that even after making the adjustments labeled (A)-(F) in chapter 3, our generated protocols do still sometimes livelock. The second is that the MI protocol is significantly slower than MSI and MESI in all workloads, which matches our expectation. The third is that, somewhat surprisingly, MESI is not in general a gain in performance compared to MSI, and in fact slows down execution considerably in some cases.

4.3 Case Study 2: Handshakes and Stalls in Unordered MSI

4.3.1 Problem Statement

In the previous case study, we looked at protocols where all the networks are ordered; that is, messages arrive in the same order as they were put on the network. In real systems, such ordered networks are often avoided, as it may be cheaper and more efficient to instead use unordered networks. Unordered networks, however, introduce additional races on the interconnects, which protocols typically deal with by adding *handshake* messages – extra messages which do not carry data, but only serve to acknowledge that a response or forwarded request has been received. Handshakes thus makes the chain of messages for a transaction longer, and more messages per transaction means more stalls in the protocol, since there is more time for conflicting requests to appear. Stalls are typically thought of as expensive operations, since they limit the concurrency of the system. Thus, conventional wisdom dictates that handshakes will slow down the system considerably.

In this case study, we explore whether handshakes really do slow down (small) sys-

tems.

4.3.2 Experiments

We use the same system model as in the previous case study (Figure 4.1), and target the same set of benchmark workloads. However, this time we look at two different implementations of fully unordered MSI protocols: one typical implementation which uses handshakes, and another implementation (proposed by Nicolai Oswald) which avoids the need to handshake forwarded requests by cleverly using more networks than the typical requests/response/forward split. Furthermore, for each of these two protocols, we consider two distinct ways of handling stalls. The first is to use so-called *z-stalls*; this means that when a message is stalled, it is simply left in place and no action is taken. Other messages arriving later on the same network must thus wait for the stalling transaction to finish, even if they are to non-conflicting addresses and could be executed safely right now.[2] The other alternative is to use *recycling*; when a message is stalled, it is removed from the in-port queue, but then re-placed on it after some fixed amount of cycles. The proposed advantage of this scheme is that messages to other addresses may go ahead in the meantime. By trying both of these methods, we can investigate whether the expected performance drop due to handshakes can be offset by recycling.

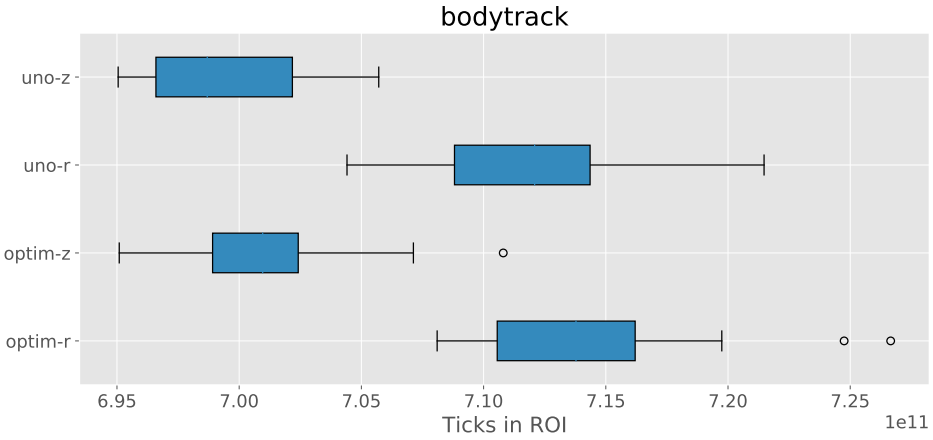
For our experiments, we use a recycling latency of 10 cycles, which is the default in Gem5 v20.1. We leave all other parameters as in Appendix A. However, given that we this time expect smaller differences in performance than before, and therefore wish to be more careful to reduce the variance in our results, we double the number of runs per set up from 10 to 20. Thus, since we have two protocols and two types of stalling to consider for each protocol, as well as 5 workloads which we want to run 20 times each, we get a total of $2 \cdot 2 \cdot 5 \cdot 20 = 400$ runs. As before, we limit each run to a maximum wall clock time of 6 hours.

4.3.3 Results & Discussion

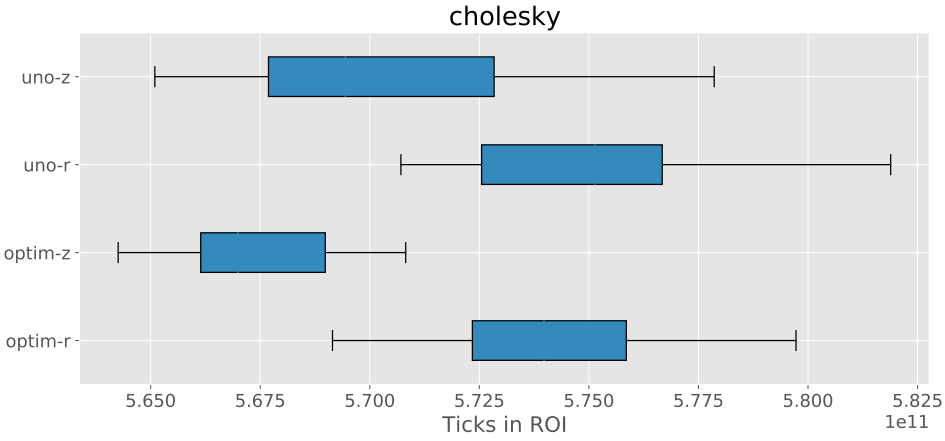
We find that 31 out of 400 runs do not finish within 6 hours, which is a comparable rate to that in the previous set of experiments. Figure 4.3 summarizes the runtimes of the 369 runs which did finish, which reveal two things. First of all, they show that the performance of the non-handshaking ”optimized” protocol is nigh equal to that of the handshaking protocol; for cholesky and swaptions, the mean run-time is a bit lower, but the distributions still overlap significantly. This contradicts our prior belief that handshakes cause frequent, costly stalls in protocols. Secondly, in every workload except swaptions, recycling actually appears to be slower than z-stalling; this is especially clear in bodytrack and cholesky, and holds for both of the protocols.

The cause of this second finding is not immediately obvious. Intuitively, recycling should increase throughput in the coherence protocol. How could it then degrade performance? One possible explanation is that since our system is comparatively small,

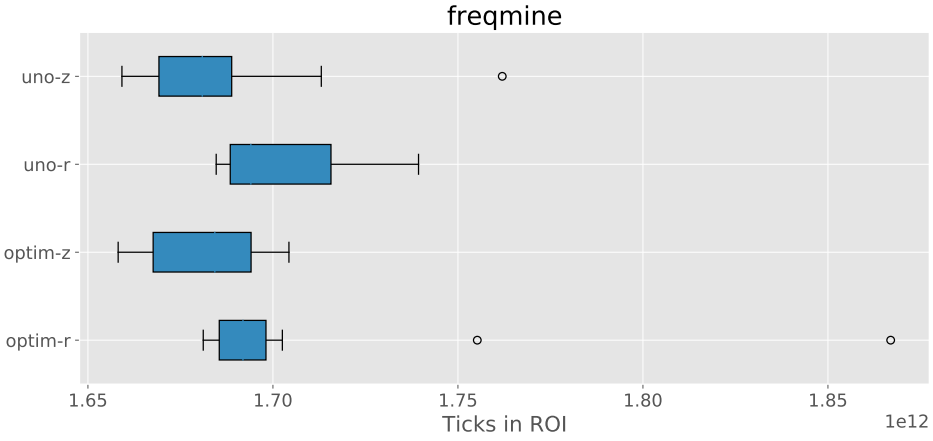
[2]Note that this implies that, in order to avoid deadlocks, the message which the machine is waiting for must not be arriving on the same port as is being stalled.



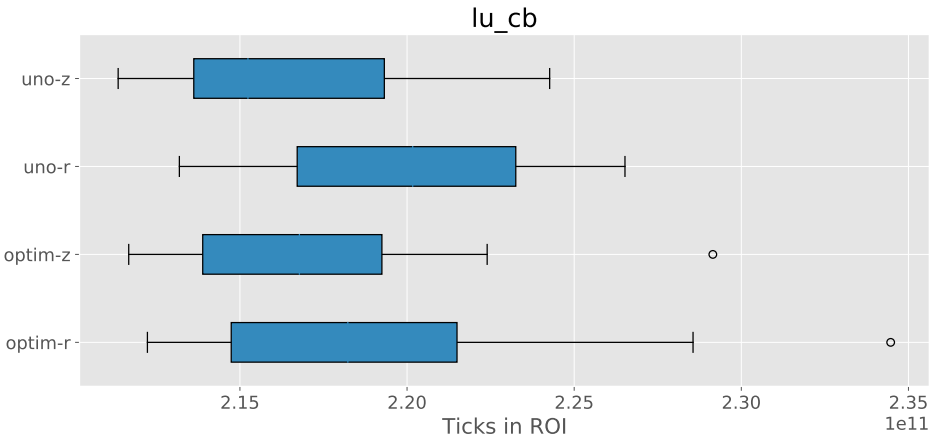
(a) Run-times in ticks for the bodytrack workload.



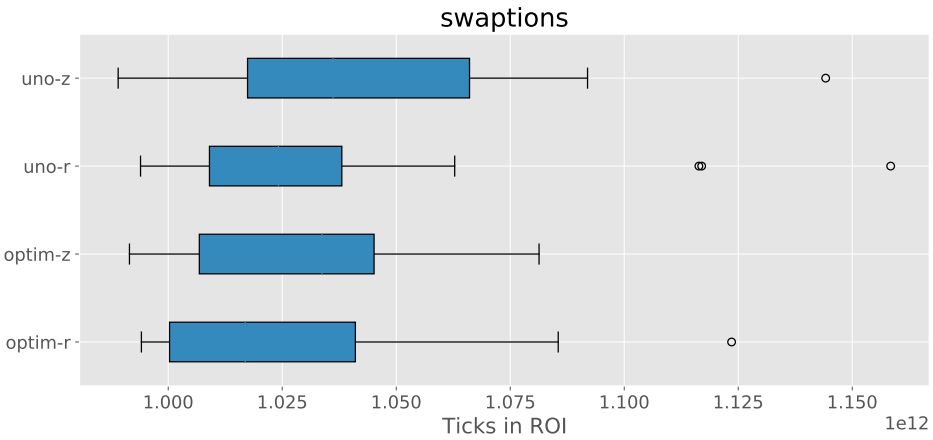
(b) Run-times in ticks for the cholesky workload.



(c) Run-times in ticks for the freqmine workload.



(d) Run-times in ticks for the lu_cb workload.



(e) Run-times in ticks for the swaptions workload.

Figure 4.3: Runtimes within each workload's ROI per protocol considered in section 4.3. Legend: "uno" is the unordered MSI protocol with handshaking, "optim" is the (attempted) optimization protocol which does not use handshaking; -z indicates a z-stall, while -r indicates stalled messages are recycled with a latency of 10 cycles.

with a maximum of two hops between the cores, z-stalls are unlikely to last for long. The fixed penalty of 10 cycles which transactions suffer each time a message is recycled may thus simple be too costly in comparison, leading to this counter-intuitive result.

4.3.4 Conclusion

In conclusion, we once again find that the generated protocols do still sometimes livelock, which supports our findings from the previous case study. When runs do not fail due to livelock, we find that our results contradict conventional wisdom. In our simple 2x2 mesh-style system, we find that stalls due to handshaking are not a source of performance degradation. Furthermore, even if they were, we find that attempting to minimize the impact of each stall by using recycling instead of simple z-stalling actually degrades performance. Thus, we conclude that handshakes are not necessarily a cause for concern, and that it may sometimes be beneficial to accept the small cost of a z-stall instead of taking (course-grained and poorly calibrated) countermeasures.

Chapter 5

Conclusion

5.1 Summary

This dissertation has presented an extension of the ProtoGen software (Oswald et al., 2018, 2020) in the form of a new backend which gives the user the option to generate cache controllers suitable for use with the Gem5 system simulator (Lowe-Power et al., 2020). Thus, the work presented herein has extended the reach of the ProtoGen tool chain, allowing it to also be useful as a way of quickly evaluating the *performance* of different designs, rather than only their correctness.

Chapter 3 discussed the conceptual challenges involved in making such a SLICC backend for ProtoGen, as well as its implementation. It laid out solutions to the biggest challenges, which were often due to mismatches between ProtoGen's DSL and SLICC both in terms of syntax and in terms of expressiveness. Then, it touched upon how, even with this preparation, several other issues came up during the implementation. Finally, it discussed remaining limitations of the backend, and how they might affect its use.

Experimental evaluation in chapter 4 showed that the generated protocols were capable of their intended use, which was to support Gem5's full-system simulation mode. However, it also revealed that the generated protocols did sometimes cause livelocks in the system when more complex workloads were run, which occurred in a total of 50 out of the 550 test runs from our two case studies (section 4.2, section 4.3). When the protocols did not cause livelocks, they were capable of generating the performance statistics necessary to evaluate the protocols. Thus, the usefulness of the proposed backend was established.

5.2 Reflection: What Took So Long?

The majority of what was discussed in chapter 3 actually took place during a small minority of the project's timeline. Thanks to the author being previously experienced with the ProtoGen codebase, initial progress was very quick; the first implementation of the backend, with the generated protocols passing the most basic of tests, was

written during a three-week period in June-July 2020. A week or two later, the issue discussed in section 3.2.1.1 had been identified and fixed. Ridiculous as it may sound, it then took until mid-November – about 5 months of continuous effort – to identify and fix the issue discussed in section 3.2.1.2, since debugging proved very difficult. Since no explicit error was thrown, there was no error trace to step back through. Even if such a trace had been available, it would have been so long that sorting through it would have been impossible; even simple SE-mode tests containing several orders of magnitude fewer instructions than booting a Linux kernel does can generate several MB worth of traces. Instead, to debug this issue we had to rely on comparing the generated protocols to sample protocols supplied with Gem5. This was interweaved with manual adjustments being made to protocols in order to understand what ”strange things” happen when you mess with how SLICC expects the protocols to be written, and how much one can mess with it while still maintaining a working implementation. The signal to noise in both of these approaches proved to be very, very low, which caused what was really a small issue to take so long to debug (despite our best efforts).

Another challenge this project faced was that the wider ProtoGen codebase was still under active development. By the time the above issue had finally been fixed, the development version of ProtoGen had undergone substantial changes in order to support future research on hierarchical and heterogeneous protocols. This included significant changes to the frontend, completely changing ProtoGen's internal representation of the protocols. In order to support the latest version of ProtoGen, the call was made to re-implement the backend, which once again took a few weeks since they changes were so substantial that the backend practically had to be rebuilt from scratch.

It is true that some of these delays would have been hard to avoid or even foresee – for example, it would have been hard to estimate how much the lack of proper documentation for Gem5 would slow down debugging. However, the project certainly would have benefited from better planning. In particular:

- Spending even more time getting familiar with SLICC's limitations early on in the project could have saved significant time later on when debugging
- Better communication with others working on ProtoGen's codebase might have avoided the need to re-write the backend from scratch halfway through the academic year
- Devising a thorough testing infrastructure for the protocols in advance might have lead to some issues becoming obvious earlier on in the process, and in ways that made their causes clearer

5.3 Future Work

This dissertation has only scratched the surface of what is possible to do with ProtoGen. Out of all the possible directions of future work, the ones we find to be most promising are:

- **Using the backend for a large-scale design space exploration.** While carrying out the case studies in chapter 4, we were genuinely surprised to find that the

results sometimes contradicted our intuition. However, it is important to remember that these two case studies only looked at five different workloads and one particular system model, which was necessary in order to fit them within the time constraints of this project (recall that we allocated each of the 550 runs a max time limit of 6 hours, meaning the case studies alone took a total of $6 \cdot 550 = 3300$ CPU hours to carry out). It may very well be that conventional wisdom, such as the advantage of MESI over MSI or that of avoiding z-stalls, holds in other settings, such as for other workloads than those considered here or for larger systems. A systematic evaluation of this would be a nice addition to the literature, and this backend facilitates precisely such a study.

- **Making more backends, such as for hardware design languages**. Recall that Hemiola (Choi, 2021) is closely integrated with Kami (Choi et al., 2017), making it trivial to synthesize real hardware from Hemiola protocols. Adding more backends to ProtoGen, for example targeting Kami, would extend the reach of ProtoGen even further and would thus help cement it as the definite tool for cache coherence protocol development.
- **Supporting hierarchical protocols.** The latest release of ProtoGen, dubbed *HieraGen*, is capable of generating hierarchical protocols (Oswald et al., 2020). This makes the software much more useful in practice, since few real systems use only one level of coherent caches. Supporting HieraGen fell outside the scope of this project, but when the backend was rewritten it was done in a matter such that adding support for hierarchical protocols in the future would be simple.

Bibliography

Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language support for writing memory coherence protocols. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, page 237–248, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917952. doi: 10.1145/231379.231430. URL https://doi.org/10.1145/231379.231430.

Joonwon Choi. *Structural Design and Proof of Hierarchical Cache-Coherence Protocols*. PhD thesis, Massachusetts Institute of Technology, 2021.

Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, and Adam Chlipala. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–30, 2017.

David L Dill. The mur ϕ verification system. In *International Conference on Computer Aided Verification*, pages 390–393. Springer, 1996.

M. Elver, C. J. Banks, P. Jackson, and V. Nagarajan. VerC3: A library for explicit state synthesis of concurrent systems. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1381–1386, 2018.

gem5.org. gem5 Resources. Resource: Linux Kernel 5.4.49. https://gem5.googlesource.com/public/gem5-resources/+/refs/heads/stable/src/linux-kernel/, n.d.a. Git repository at revision 'v20.1.0.5'. Accessed February 2020.

gem5.org. gem5-20 Working Status of Benchmarks. https://www.gem5.org/documentation/benchmark_status/gem5-20, n.d.b. Accessed February 2020.

Jason Lowe-Power. Learning gem5, part iii: Modeling cache coherence with ruby, Sep

2017. URL http://learning.gem5.org/book/part3/index.html. Accessed June 2020.

Jason Lowe-Power. Email communication, URL https://lists.gem5.org/archives/list/gem5-users@gem5.org/message/3X3RDMQP4NOQVD5QQM3TTUDDJ426WCVB/, July 2020.

Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, et al. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

Vijay Nagarajan, Daniel Sorin, Mark Hill, and David Wood. *A Primer on Memory Consistency and Cache Coherence*, volume 15. Morgan & Claypool Publishers, 2nd edition, 02 2020. doi: 10.2200/S00962ED2V01Y201910CAC049.

Theo Olausson. Towards the automatic synthesis of cache coherence protocols. *Master of Informatics Project (Part 1), University of Edinburgh*, April 2020.

Nicolai Oswald, Vijay Nagarajan, and Daniel J Sorin. Protogen: Automatically generating directory cache coherence protocols from atomic specifications. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 247–260. IEEE, 2018.

Nicolai Oswald, Vijay Nagarajan, and Daniel J Sorin. Hieragen: automated generation of concurrent, hierarchical cache coherence protocols. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 888–899. IEEE, 2020.

Mark S Papamarcos and Janak H Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th annual international symposium on Computer architecture*, pages 348–354, 1984.

Mahyar Samani, Ciro Santilli, and Saverio Miroddi. Parsec benchmark, n.d. URL https://github.com/darchr/parsec-benchmark. Retrieved February 2021.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. TRANSIT: Specifying protocols with concolic snippets. *SIGPLAN Not.*, 48(6):287–296, June 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462174. URL https://doi.org/10.1145/2499370.2462174.

Patrick Valduriez. Shared-memory architecture. In LING LIU and M. TAMER ÖZSU, editors, *Encyclopedia of Database Systems*, pages 2638–2638. Springer US, Boston, MA, 2009. ISBN 978-0-387-39940-9. doi: 10.1007/978-0-387-39940-9_1082. URL https://doi.org/10.1007/978-0-387-39940-9_1082.

Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news*, 23(2):24–36, 1995.

Appendix A

Detailed Settings for Case Study Experiments

In this appendix, we give a few more details of the experimental set up used in the case studies. The values of response and request latencies were copied from example protocols supplied with Gem5. Parameters not mentioned here were set to their default values (as of Gem5 v20.1).

Component	Settings
L1 Cache(s)	size = 64kB; associativity = 2; response latency (number of cycles before a response message is added to the network) = 1
Directory	response latency = 6; memory request latency = 1 (number of cycles before a memory request is sent to the memory controller)
Network	network type = garnet2.0; router latency = 1; link latency = 1; link width = 128b; routing algorithm = XY