ProtoGen-MLIR: an MLIR Compiler for Cache Coherence Protocols

Petr Vesely

MInf Project (Part 1) Report

Master of Informatics
School of Informatics
University of Edinburgh

2021

Abstract

This paper presents ProtoGen-MLIR, which is an MLIR compiler for compiling and optimizing coherence protocols from their stable declarations. Coherence protocols as notoriously difficult to reason about, especially when presented as highly concurrent implementations with dozens of transient states. This motivated ProtoGen-MLIR, which defines a compiler for a high-level Domain-Specific Language (DSL) named PCC. Using this DSL we can specify a coherence protocol with atomic transactions between stable states. Motivated by the finding from the original ProtoGen, ProtoGen-MLIR uses these stable specifications to generate a concurrent protocol that removes the need for atomic transactions. ProtoGen-MLIR also allows all generated protocols to be verified formally as the compiler targets Murphi code.

Acknowledgements

Firstly I would like to thank my supervisors Dr. Vijay Nagarajan & Dr. Tobias Grosser. Thank you for your excellent mentorship, you wealth of knowledge and constant support, guidance and encouragement throughout this project. At points I had doubted if it was even possible and you always helped me through, thank you.

Additionally I would like to thank Nicolai Oswald & Vasilis Gavrielatos, for begin so willing to share your knowledge and experience for the background that motivated this project.

Lastly I wish to thank my family and my friends, for being there and supporting me throughout this journey.

Table of Contents

Chapter 1

Introduction

1.1 Overview

Designing custom silicon is hard. Designs and prototypes must be extensively tested to ensure product quality, and often this verification becomes the bottleneck in chip design [4]. Catching defects late in development or after production can be prohibitively expensive, due in part to redesigning of the architecture; being beaten to market through delays; and possibly even product recalls due to poor quality. As such the industry employs a variety of techniques to verify designs, primarily through simulation, but some components can also be verified formally. Often hardware is *programmed* through a Hardware Description Language (HDL) such as Verilog, which can then be converted to a satisfiability problem and verified formally [4]. However, such techniques result in considering a very large state-space, where only a fraction of the possible states are necessary for verification.

The rate of semiconductor technology has been slowing for several years, resulting in an industry wide shift for more specialised architectures, generally called accelerators. Such accelerators are generally designed for specific workloads in mind such as Machine Learning. Therefore, they are often implemented as custom silicon as they do not require features of general purpose computing architectures and instead implement specific optimizations for their designed workload. With this trend continuing, it will be necessary for even small manufacturers to create custom silicon to accelerate their application requirements.

One approach to tackle this problem is to define a set of domain specific languages in which the designer can specify a hardware description at a high-level and use a compiler to generate an equivalent low-level description such as Verilog, which can then be implemented and verified. Programming languages such as C already employ such a strategy, by providing a high-level abstraction over assembly. The programmer then no longer has to concern themselves with registers and machine instructions, and instead uses the language to define the behaviour of the program. This allows them to implement code faster, with fewer errors and reduces mental overhead for the programmer. Furthermore, as the compiler becomes more advanced, optimizations can be added that result in faster executing code, and higher-quality code than if attempted to

be written by hand. Why can't we do the same for hardware descriptions? Suppose we can specify, for example, a processor pipeline as a high-level abstraction using a domain specific language, and then compile this description. The compiler can then apply optimizations, verify the design and generate the output target. Were this to exist, it would allow much smaller and specialised teams to develop custom silicon for accelerators, as the verification and design stages become far simpler. Moreover, the generated output is correct-by-construction, which could allow the designers to implement much more complex systems that would otherwise be impossible to verify with traditional methods.

This project focuses on cache coherence protocols in modern multi-core CPUs, however cache coherence protocols can be deployed in any architecture with private caches and a shared memory interface. On the surface, cache coherence protocols are deceptively simple. Their operation can often be described using only a handful of states and their coherence mechanisms can easily be reasoned about. However, such descriptions assume *Atomic Transactions*, where only a single coherence transaction is happening at any one time, which is almost never the case in actual hardware implementations. Relaxing atomic transactions greatly complicates the coherence protocol, by introducing many more transient states, that encode the progress of a transaction as well as the progress of other transactions that are occurring simultaneously. For example, a simple MSI protocol, with only three stable states, results in a protocol with eighteen total states, when allowing for non atomic transactions [8].

1.2 Aims

This project attempts to design an MLIR compiler for a high-level Domain Specific Language (DSL) used to specify cache coherence protocols. Fundamentally, this project presents a general intermediate representation (IR) for cache coherence protocols, that will allow optimizations to be implemented by directly manipulating and extending this representation. As a result this will produce an optimized version of the provided stable protocol, which considers and handles racing transactions. After these optimizations have been implemented, as part of the intermediate representation, the compiler targets a Murphi backend to verify the correctness of the generated protocol.

As a consequence this would allow the designer of a cache coherence protocol, to reason about a protocol using the *textbook* definitions i.e. without considering concurrency and racing transactions. This Stable State Protocol (SSP) and any necessary hardware specifics, such as the interconnects, are specified at a high-level using the DSL. The compiler uses the definitions from the DSL and translates them into the general intermediate representation to perform the optimizations. This is important as creating the equivalent protocol by hand is difficult and error prone, therefore using such a tool will provide *correct-by-construction* protocols, further aiding in verification efforts. Moreover, the correctness of the generated protocols can be verified formally with Murphi. In practice, coherence designers could iterate their designs rapidly, with instant verification, allowing them to quickly obtain a highly-efficient coherence protocol for little development cost.

1.3 Previous Work

1.3.1 Circt

Circt is an existing open-source project to address the lack of open source tooling for Electronic Design, including processor architecture [6]. The project provides the user with a modular and reusable library of high-level intermediate representations which, through their custom optimizations and pipelines, can produce industry standard output targets such as Verilog. The project provides many higher level abstractions for designers, and applies custom domain specific optimizations before lowering to their desired target such as Verilog. The project leverages MLIR for its compiler infrastructure, and the success shown in the project for application to hardware design, was a big motivation in attempting to model cache coherence using MLIR.

1.3.2 ProtoGen

ProtoGen is a tool for generating highly concurrent cache coherence protocols from their stable state definitions and is the primary motivation behind this project. ProtoGen defines a DSL for specifying the architecture and behaviour of the cache and directory controllers at a high level, without considering concurrency [8]. In fact, a subset of the language is used as the input to the compiler presented. ProtoGen performs highly specialised optimizations by applying domain knowledge of the architecture, and generating the necessary transient states and transitions to create an equivalent, but highly concurrent protocol. A lot of the same reasoning about racing transactions and adding additional concurrency is based on the findings from ProtoGen, however Protogen-MLIR uses an overall different architecture and approach by augmenting an IR, motivated by the findings of Circt.

1.3.3 Teapot

Teapot is a domain specific language and compiler for specifying the behaviour of cache coherence protocols [1]. However, unlike ProtoGen or ProtoGen-MLIR, Teapot does not provide any optimizations in terms of additional concurrency. Teapot instead provides a high level programming language to specify the behaviour of protocols using functional programming concepts. Specifically, Teapot leverages *continuations* to track the execution of a coherence transaction, by allowing cache or directory controllers to *yield* their execution and be resumed later. This is particularly elegant in transactions which send a request and `await` some response.

Teapot code is compiled to C, but also includes a Murphi backend for formal verification, similarly to ProtoGen and ProtoGen-MLIR. Teapot continuations were analysed in detail to discover if their application could be directly applied to Protogen-MLIR, however we found no use for them in this project. ProtoGen-MLIR requires additional information about its own state as well as the state of the directory to make reasoned decisions about which forwarded messages it must handle (for optimizations), which is not possible through continuations. Moreover, our DSL provides an elegant `await` syntax for stalling the execution of a transaction.

1.4 Contributions

As part of this effort we present ProtoGen-MLIR, which is an MLIR based compiler that has the following key components.

- Parser and MLIR frontend for our custom Domain-Specific Language
- MLIR Optimization pipeline, which transforms our stable protocol to a concurrent version.
- Murphi backend which generates valid Murphi code for final protocol verification

Chapter 2

Background

In this chapter we introduce the required background for ProtoGen-MLIR. In section 2.1 we introduce the general problem of cache coherence, followed by the baseline architecture and the challenges this presents and how they can be tackled. In section 2.2 we present the function of directory based protocols and introduce the class of MOESI protocols used extensively throughout this project. Next we present MLIR in section 2.3 and discuss its advantages for implementing ProtGen-MLIR, before finally presenting Murphi in section 2.4 which provides the ability to formally verify our generated protocols.

2.1 Cache Coherence

Most modern CPU's are multi-core, meaning that a CPU can execute threads in parallel by allowing each thread to run on a separate core. Moreover, most systems provide a shared memory interface, meaning threads running in parallel can read and crucially write to memory used by another thread. Furthermore, modern systems rely heavily on caching, which stores frequently accessed memory addresses on a fast-access chip physically close to each CPU core. Caching is important for performance as memory accesses are not randomly distributed, rather they exhibit *temporal locality* (recently accessed memory addresses are likely to be accessed again) and *spacial locality* (memory addresses close to the a recent access are also highly likely to be accessed) [10]. By storing recently accessed memory addresses in a fast-access cache, the CPU will hit the cache with a high probability, thus incurring a significant performance increase as reading from memory is many times slower. In Figure 2.1 we present the baseline architecture considered throughout this project, we can see that all cores have access to a shared main memory, but each core has a private data cache. Loads and stores from the core's pipeline are issued to the cache controller, which then communicates with other caches to synchronize and fulfil the request to the core. Cache and directory controllers communicate through an interconnect network that interconnect all the controllers. We do not consider any specific hardware implementation as this is not required for reasoning about the protocol. We leave such hardware decisions to the implementation.

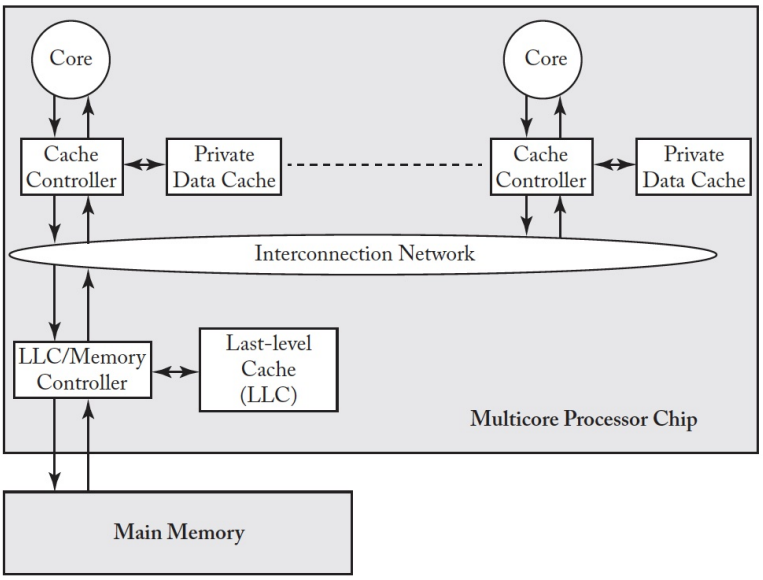

Figure 2.1: Baseline System Model [7]

With such a system it is easy to see how cache incoherence would quickly arise. For example, suppose we have a multi-core CPU with two cores (C1 & C2) each of which have a private data cache and both are accessing the same memory address. When C1 & C2 read this address they both make a copy and store the result to their private data cache. C1 now performs a write operation, and the result is written back to main memory. However, now if C2 performs a read on the address it will hit the private data cache which contains the unmodified data, because the modified data was not updated in the private data cache when C1 performed the write operation.

For *coherence* to be maintained, meaning that every core holds the most up-to-date value in their private data cache and avoid the situation described previously, it is a requirement for cache coherence protocols to ensure *Write Propagation* and *Transaction Serialization* [7]. Write Propagation, ensures that any writes to a memory address are propagated to every copy held in a private cache, meaning that when C1 performed the write C2 would would have it's copy updated to reflect this new value. Transaction Serialization, ensures that all reads/writes are seen by all cores in the same order, i.e. there is a total ordering of transactions observed by all cores. This is necessary, because suppose that C1 and C2 each performed writes to the same address, but each observe their write operation to happen first. This would clearly result in incoherence as C1 would observe the write from C2, and C2 would observe the write from C1 in their respective caches.

For implementation and reasoning about protocols with these constraints, it is more useful to consider invariants, which when maintained ensure Write Propagation and Transaction Serialization, and thus coherence. Consider the following invariants:

- *Single-Writer-Multiple-Reader (SWMR)* : At any point in time any memory location can have either a single actor with write permissions (Single-Writer), or many actors with read permissions (Multiple-Reader)[7]

- *Data-Value Invariant* : This ensures that the last write operation is reflected in the next read operation.[7]

When these two invariants are maintained it can be shown that this will enforce Write Propagation and Transaction Serialization and thus will result in a protocol that maintains coherence [7]. If we can then prove, for example through a model checker like Murphi, that these two invariants hold for every possible state of the cache, we can be confident that the coherence protocol is correct.

Real-world protocols implement these invariants by assigning each memory block a state which has some associated permission (read/write/none). If a core wishes to perform an operation on the block (i.e. load or store), the cache controller must first obtain the block with the required permissions before performing the operation. Typically, caches can request permissions through a *Coherence Transaction* in which requests are sent to and responses received from the directory or other caches to obtain the block with the required permissions. However, a well designed protocol must also ensure caches yield their permissions correctly to allow other caches access to the same cache block. Obviously, adding such a protocol incurs some performance penalty, so a coherence protocol must also be well designed to ensure the overhead is minimised as much as possible. All this combined shows the complexity and erroneous nature of designing a correct, efficient and performant coherence protocol.

2.2 Directory Coherence Protocols

Although coherence protocols can be implemented using multiple techniques, directory based protocols are the most common and are therefore the only type considered in this project (see Figure 2.1). A directory based protocol consists of two different controller types: **directory controllers** and **cache controllers**. Each controller maintains the state for every memory block and updates it based on the rules defined by the coherence protocol. Therefore each controller can be implemented as a Finite State Machine (FSM) that transitions blocks between stated according to the rules defined by the protocol.

A group of standard coherence protocols, known as the MOESI protocols (often pronounced either “MO-sey” or “mo-EE-see” [7]), utilise a combination of **M**odified, **O**wned, **E**xclusive, **S**hared and **I**nvalid states, to model cache blocks with different permissions. These states are known as stable states, and represent what permissions the cache has. For example S(hared) state allows the cache to read the data, however for writes the cache needs to first obtain M(odified) state by issuing a coherence transaction. A transaction typically consists of multiple steps, for example a transaction from an I(nvalid) state to a S(hared) state, involves a request to the directory for shared state and a corresponding response. After sending the initial request, the cache controller must transition to some transient state to represent what has occurred, and only once the corresponding response arrives can it transition to the desired stable state. The directory controller also uses these same stable states, however they are used to represent the current state of the caches. For example: if the directory is in state M, this means that some cache currently has *write* permissions and if directory is in state

I, then no cache is currently caching the block.

2.3 MLIR

MLIR (Multi-Level Intermediate Representation) is a novel approach to building reusable and extensible compiler infrastructure. In the past, each language implemented it's own compiler, however recently lots of compilers now target LLVM instead of general machine code, simplifying the compilation and leveraging the powerful optimizations LLVM provides. However, there is no one-size-fits-all, and often higher level optimizations cannot be realised with a low-level IR such as LLVM-IR. Instead compilers implement a custom frontend with higher-level representations to implement these optimizations or checks before *lowering* to LLVM-IR. Rust, for example, is a complex language with a very strict type system and memory safety requirements, and compiles through two intermediate representations before finally targeting LLVM-IR for this very reason.

MLIR addresses this by providing a declarative way to define custom representations at different levels of abstraction and semantics, unlike the fixed instruction set of LLVM-IR. Furthermore it provides a unified surrounding infrastructure for validation, conversion between representations and optimizations. This allows for a full compilation from high-level IR through to LLVM-IR, without requiring engineering time to custom frontend implementations.

MLIR is particularly useful for domain specific languages, which still remain a challenge to implement. Furthermore, due to their nature, certain aspects of the compiler are often overlooked or perform inadequately. This can cause downstream problems like slow compilation, bugs in the code, mediocre debugging and a generally poor user experience. MLIR provides a robust supporting infrastructure, and extensible instruction set, making it much simpler for language designers to create highly efficient, maintainable and quality compilers easily. With general purpose computing slowly moving to become ever more specialised, it is expected that highly optimised DSLs will be the norm for each specialization. MLIR is an important and necessary technology to allows the language designers to create such compilers [5].

MLIR is used extensively in this project to model the PCC DSL through MLIR Operations. Using the supporting infrastructure of MLIR allows ProtoGen-MLIR to implement optimizations by directly inspecting and modifying the IR. Rewrites of the IR are defined declaratively by defining when a optimization can be applied, and the specific rules on how to perform the rewrite. This makes it easy to debug, as MLIR makes it clear exactly what matches and rewrites have been performed.

2.4 Murphi

Murphi (sometimes confusingly written as MurΦ and pronounced *Murphy*) is a model checker and programming language developed at the University of Utah that is specifically designed to test cache coherence protocols [2]. It exhaustively searches through

all possible states and checks for deadlock freedom, and any other invariants we wish to maintain, like SWMR. Murphi was chosen as the compilation target for ProtoGen-MLIR, allowing the optimized protocol to be verified formally.

The Murphi Reference Manual[3] details how a Murphi program is verified

> Repeat forever:
>
> a) Find all rules whose conditions are true in the current state. (i.e. conditional expressions are true, given the current values of the global variables).
>
> b) Choose one arbitrarily and execute the action, yielding a new state.

However, Murphi does not apply these rules naively, instead it relies heavily on symmetry to reduce the state space. For example, Murphi defines a `scalarset` data type that allows Murphi to consider every element of the set as symmetrical. We can use this to define all caches in our protocol as members of a `scalarset` allowing Murphi to significantly reduce the state space. If C_1 is in state S_1 and C_2 is in state S_2, then Murphi will consider this equivalent to the case of C_1 in state S_2 and C_2 in state S_1. Furthermore, Murphi does not allow generic integer types, instead allowing the programmer to only specify integers within finite ranges, again to attempt to limit the state space that must be explored.

In Section 3.7, we describe in detail the process of generating valid Murphi code from the optimized MLIR Operations generated from the compiler, allowing us to formally verify generated protocols.

Chapter 3

Designing the MLIR Compiler

In this chapter we present the design, implementation specifics and optimizations of the ProtoGen-MLIR compiler. We introduce the input DSL and its support for specifying directory-based coherence protocols. Additionally, we present the dialects implemented in MLIR to model the different abstractions of a cache coherence protocol, and detail the process of implementing optimizations that introduce additional concurrency for increasing protocol performance. Finally, we present the Murphi backend to generate valid Murphi code for verification.

Throughout this chapter we make reference to the ProtoGen-MLIR pipeline, which consists of the following stages, each of which is presented in detail.

- Remove `await` structures by splitting into transient states
- Insert additional Mutex operations to allow for Atomic Transaction Verification
- Apply the ProtoGen Optimizations
- Convert all Operations to the Murphi Dialect

3.1 The PCC DSL

ProtoGen-MLIR borrows the DSL from the original ProtoGen, named PCC (ProtoGen Cache Coherence). This was chosen as it was well understood and provided all the necessary language support for specifying the required components of a cache coherence protocol. Moreover, designing a custom DSL was out of scope for this project. Furthermore, the DSL already provides an elegant way of specifying the architecture and behaviour for the controllers so there was little benefit in a redesign. ProtoGen-MLIR uses a subset of the language to support the MI, MSI and MESI protocols only. The MI protocol is the most basic MOESI protocol that uses only two states: **M**odified and **I**nvalid. When, a cache wishes to read or write data into its private cache it must obtain the block in state M (see Figure 3.1). The focus initially was to implement a vertical for the full MI protocol with the minimum required features to validate the baseline functionality before attempting more complex protocols such as MSI or MESI. Upon successful completion of the limited compiler for the MI protocol, we added additional

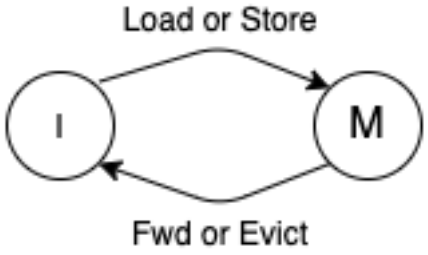

Figure 3.1: Stable State transitions for the MI Protocol

language support for integer ranges and sets and conditional branches, which allowed us to implement MSI & MESI successfully.

To specify a directory based protocol, it is necessary to store additional data along with the state. For example, when the directory has issued **M**odified state to a cache, it needs to maintain a reference to this owner, so that when the directory receives a coherence request from a different cache it can correctly forward the request to the owner. This additional state is referred to as the *auxiliary state*. Listing 3.1 shows the required cache and directory state definitions for the MI Protocol. The definition specifies that we have a set of cache controllers and a single directory controller and each must maintain their current state along with the data of the cache line. The directory controller also needs to maintain a reference to the current owner (as explained). PCC comes with some built-in data types: **State** (one of the stable states); **Data** (data from memory); **ID** (a reference to any controller, cache or directory), which can be used to define the auxiliary state.

```
Cache {
    State I;
    Data cl;
} set[NrCaches] cache;

Directory {
    State I;
    Data cl;
    ID owner;
} directory;
```

Listing 3.1: Specifying the Directory and Cache Aux State

Along with definitions of the cache and directory, PCC also provides a high level abstraction on specifying the interconnects between controllers. Controllers can be connected through a variety of interconnect technologies, however when reasoning about protocols, the only important property of the interconnect is if it enforces ordering. Listing 3.2 shows the PCC definition of the necessary networks for the MI protocol and the required ordering constraints on each. We see that even a simple protocol such as MI requires three interconnects. Naturally we have one for requests and responses which do not require ordering, however the third forward network is necessary for

when the directory forwards a request to another cache to fulfil. This network must be ordered, due to the fact that the directory acts as the serialization point in the system, and this exact ordering must be observed by the caches when receiving forwarded messages. Ordering constraints however can be relaxed on the other networks due to the fact that the directory will serialise incoming requests, and responses too can arrive out-of-order as the controller implements the logic to handle them. Relaxing ordering constraints gives more freedom to the final hardware design as unordered networks can be replaced with an ordered one, moreover unordered networks could utilize topologies that aid with final performance.

```
Network { Ordered fwd;
          Unordered resp;
          Unordered req;
          };
```

Listing 3.2: Specifying Interconnects

Finally, PCC provides an elegant interface for specifying the behaviour for each of the controllers. Listing 3.3 shows the transaction a cache must perform, when in state `I` and performing a `load`. When the cache receives a `load` in state `I`, the controller sends a `GetM` request to the directory on the `req` network. It then expects a `GetM_Ack_D` message from the directory or a cache controller to obtain the cache line and update its state to `M`. The full protocol is then defined as set of such transactions, which collectively define how the controller will behave in each situation. We can see how PCC provides the abstraction of atomic transactions through the keyword `await`, as the behaviour of the transaction assumes that it is the only transaction occurring.

```
Process(I, load, State){
    msg = Request(GetM, ID, directory.ID);
    req.send(msg);

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

Listing 3.3: Cache Load Transaction

3.2 PCC to MLIR Frontend

To enable us to implement the ProtoGen-MLIR pipeline, we first need to parse a PCC file into an Abstract Syntax Tree (AST) and generate valid MLIR Operations for input to the ProtoGen-MLIR pipeline.

We used ANTLR4 [9] to auto-generate a parser from a grammar definition of the PCC language, and construct the desired AST. ANTLR4 grammars define a set of tokens (like keywords and identifiers), and a set of rules to match against. A rule is a regular expression of tokens and ANTLR4 will match rules greedily to generate a parse tree. Running ANTLR4 with this grammar file generates classes and type declarations required to parse a PCC file. Importing these classes into the compiler, along with the ANRL4 runtime, allows us to parse a PCC file into an AST.

We then traverse this tree and emit MLIR in the PCC Dialect (see Section 3.3), which in doing so presented some key challenges. Firstly, we must perform some basic type inference on basic types such as Integer or Boolean and allocate them correctly. Complex types such as *State*, *Message* or *ID* are also allocated, however their types are opaque, meaning they are only symbolic and do not physically allocate space in memory. This is sufficient for this compiler, as we are targeting Murphi, which implements it's own compiler that will perform the physical allocations. Secondly, while parsing we need to maintain symbolic references to both global and local variables, and their generated MLIR Operations. For example, consider the statements from a PCC `Process` block shown in Listing 3.4. The first of these statements allocates a local variable `msg` and assigns to it the result from a `Resp()` message constructor. The next statement then references the global `resp` network to send `msg`. To generate MLIR Operations that correctly reference both local and global variables we maintain a data structure (not dissimilar to a hash map) to map variable identifiers to their MLIR results. The only additional challenge, was to drop all local references, when moving out of the current scope.

```
msg = Resp(GetS_Ack, ID, GetS.src, cl);
resp.send(msg);
```

Listing 3.4: PCC send message

3.3 Defining the PCC Dialect

MLIR is in nature Multi-Level, meaning there is no fixed representation and any custom representation outside of the standard dialects must be specified. Thus, to define a custom representation for the PCC DSL, we must specify an MLIR *Dialect*. A Dialect in MLIR is a collection of custom *Operations* all grouped under the same namespace. An Operation (or Op) is the fundamental entity of MLIR. Operations themselves can contain a list of regions, and regions themselves can contain a list of blocks, and blocks can again contain Operations, allowing for recursive structures. Operations also accept zero or more operands and return zero or more SSA values, which can then be used as operands by other operations. From Listing 3.5 `pcc.function` is an operation, it contains a single attached region denoted inside of the curly braces. This region contains only a single block with the nested `pcc.constant` operation, which defines a constant and returns this as an SSA value, denoted by `%0`. Furthermore, operations can also be assigned a list of attributes, these must be compile time known and cannot be generated dynamically like {`value = 3`}.

```
"pcc.function"()({
    // Region
    %0 = "pcc.constant"(){value = 3} -> i64
}): () -> ()
```

Listing 3.5: PCC Dialect Function Op

Custom dialects define the semantics of higher level constructs, for example `pcc.function` semantically defines a function and logically all operations nested within its regions belong to that function. These higher level semantic constructs are incredibly useful for implementing optimizations, before they are transformed to lower level operations and compiled to machine code. As detailed before the compile target for this project is Murphi which itself is a high level programming language, but it too will be implemented as an MLIR dialect, and conversion between these two dialects will have to occur before we can emit correct Murphi code.

```
"pcc.function"() ( {
    %6 = "pcc.msg_constr"() {msgType = "Request", params = ["GetM", "ID", "directory.ID"]} : () -> i64
    "pcc.send"(%6) {netId = "req"} : (i64) -> ()
    "pcc.await"() ( {
        "pcc.when"() ( {
            "pcc.set"() {id = "cl", value = "GetM_Ack_D.cl"} : () -> ()
            "pcc.set"() {id = "State", value = "cache_M"} : () -> ()
            "pcc.break"() : () -> ()
        }) {msgId = "GetM_Ack_D"} : () -> ()
        "pcc.await_return"() : () -> ()
    }) : () -> ()
    "pcc.return"() : () -> ()
}) {action = "store", cur_state = "cache_I", machine = "cache"} : () -> ()
```

Listing 3.6: PCC Dialect I load Transaction

Listing 3.6 shows a snippet of the generated MLIR for the PCC definition shown in Listing 3.3. This MLIR generated is semantically very similar to the language definition, which is by design as the operations that belong to the PCC dialect should match as closely as possible. Dialects like the PCC Dialect are sometimes referred to as *Interface Dialects*, as they provide and interface between the DSL and the MLIR operations. This allows PCC to be fully expressed through MLIR operations, without losing any high level information, which is necessary for implementing optimizations. Having PCC fully expressed in MLIR, allows transformations to be performed directly on this IR using MLIR's supporting infrastructure, instead of implementing this functionality through a custom front-end before emitting MLIR. After any transformations and optimizations have been performed (See Section 3.4 & 3.5), all the remaining operations must be converted to operations in the Murphi Dialect, which can then be used to emit valid Murphi (see Section 3.7).

3.4 Generating the equivalent Stable State Protocol

As a first step, we attempt to compile the protocol without introducing any additional concurrency. This is an important step as it allows us to verify the correctness of the stable protocol, while implementing the full compilation pipeline albeit without any optimizations. However, the stable state protocol is a simplistic view of a coherence protocol and therefore attempting to verify it as is will result in a deadlock. To over-

come this and verify the protocol with the atomic transaction constraint we introduce locks to prevent racing transactions from firing while another transaction is executing.

Consider the transaction in the MI protocol when the CPU wishes to read a block currently in state I. The PCC definition for this transaction is detailed in Listing 3.3, and we can see that the cache initially issues a request and awaits a response. To encode each stage of this transaction we have to introduce additional transient states. For example, instead of awaiting we instead transition to a suitable transient state that represents that we started in state I and performed a load. In Figure 3.2 we show how the transaction now proceeds: (1) after issuing the `GetM` we transition to a transient state `I_load`, which represents the state of the cache inside of the `await` block; (2) once we receive the message `GetM_Ack_D` we can now transition to the stable state `M`, completing the transaction. The transient state encodes the progress of the transaction through its name; for example the transient state introduced in Figure 3.2 `I_load` represents the position in the transaction starting from stable state `I` and having performed a `load` operation. The naming specifics become important when introducing additional concurrency in section 3.5.

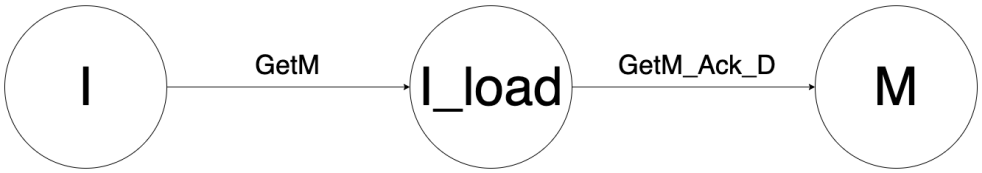

Figure 3.2: I→M Transaction with trasient state

With this example, we can reason that whenever we observe an `await` in PCC, we are effectively transitioning to a transient state that reflects its start state and the response it's expecting. In PCC `await` blocks can have many `when` guards, which in turn can have more `await` blocks, resulting in the introduction of many transient states, which emphasises how error prone designing such protocols can be when implemented manually, and we have yet to introduce any additional concurrency.

```
"pcc.function"() ( {
    %6 = "pcc.msg_constr"() {msgType = "Request", params = ["GetM", "ID", "directory.ID"]} : () -> i64
    "pcc.send"(%6) {netId = "req"} : (i64) -> ()
    "pcc.set"() {id = "State", value = "cache_I_load"} : () -> ()
    "pcc.return"() : () -> ()
}) {action = "store", cur_state = "cache_I", machine = "cache"} : () -> ()

"pcc.function"() ( {
    "pcc.set"() {id = "cl", value = "GetM_Ack_D.cl"} : () -> ()
    "pcc.set"() {id = "State", value = "cache_M"} : () -> ()
    "pcc.break"() : () -> ()
    "pcc.return"() : () -> ()
}) {action = "GetM_Ack_D", cur_state = "cache_I_load", machine = "cache" } : () -> ()
```

Listing 3.7: Separating into Transient State

Listing 3.7 shows the resulting IR after transforming the IR in Listing 3.6 by dividing the transaction into individual steps. Observe that we now have two `pcc.function` operations that effectively represent state handlers, which signal the two steps in the transaction. Further, note that we must introduce an additional `pcc.set` operation to the original function to transition to the transient state.

As mentioned, we must to maintain a Mutex for every cache block and acquire this Mutex before executing any transaction. Transactions in PCC begin with a `Process`

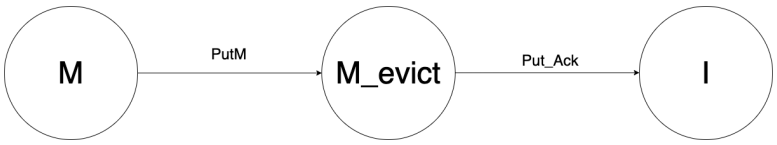

Figure 3.3: MI evict transaction

block and terminate with a `break` statement, therefore the addition of *aquire* and *release* Mutex operations can easily be added directly to the IR, using MLIR's supporting infrastructure.

3.5 Performing Optimizations

In Section 3.4 we show how we can generate the necessary transient states to encode the steps of a coherence transaction, however we must introduce locks to ensure that *Atomic Transactions* are maintained and we can verify the protocol. However, to be able to relax the atomic transaction constraint we must be able to handle multiple concurrent transactions for each block. In essence, we must answer the question: How should a cache controller respond when it receives a forwarded request from the directory or another cache, related to a concurrently executing transaction for the same block? For the cache to react to a concurrently executing transaction we must first understand the order that requests arrived at the directory.

In a directory based protocol, the directory acts as the *serialization point* for all requests; even if two requests arrive simultaneously, the directory will break the tie and order these requests sequentially. Because of this, caches are able to deduce the order in which requests are serialized at the directory based on the forwarded responses they receive.

For example, consider the transaction in the MI Protocol, where a cache wishes to evict a block in state M. The cache first issues a `PutM` request to the directory and once it receives a `Put_Ack` response can it transition to state `I` (see Figure 3.3). This is the *happy* path as no other competing transactions. However, suppose that while the cache is in the transient state `M_evict` it receives a forwarded `GetM` request from the directory. The cache can now deduce the fact that another `GetM` request was ordered at the directory before its own `PutM` causing the directory to forward the request. Using this insight we can always deduce the ordering of messages received at the directory, relative to our own, i.e. if it was ordered before or after. By knowing the ordering of message at the directory, we can handle these forwarded messages and allow transactions for the same block to occur concurrently.

To be able able to handle forwarded requests, we must first know which forwarded requests could *potentially* arrive for any transient state. Transactions logically occur between stable states: i.e. I→M or M→I, even though transient states are involved. This implies that during any transient state in the transaction the directory will *see* the cache as being in either one of these stable states. We say that each transient state also has a **logical start state** and a **logical end state**, which represent the two potential states that the directory can *see* the cache in. If the cache *sees* a forwarded request, and

recognises that the directory must have only sent that message because the directory *sees* the cache in either the logical start or end state, then the cache can correctly handle this request by assuming it is in that logical state.

Knowing this fact, how can we then determine which requests could potentially arrive & how do we determine the logical start and end states for each transient state? Sometimes determining the logical end state of a transient state cannot be fully realised, due to branches that can occur is certain protocols. For example in the MESI protocol, the E(xclusive) state is issued when they're no other sharers, otherwise S(hared) state is issued, therefore this transient state that represents awaiting the response from the directory cannot deterministically know if the logical end state is S or E. A solution to this could be to track every possible logical end state and consider every situation individually, however such an implementation would require additional analysis of the protocol and a complex redesign of the current implementation. Furthermore, not considering the end state can be done successfully without incurring deadlocks at the cost of potentially not introducing additional concurrency (discussed later).

How then can we determine the logical start and potentially logical end state(s) for a transient state? Firstly, the logical start state is encoded into the state name for each transient state. For example, in Figure 3.3 we introduced a transient state `M_evict`, which represents that that the cache is still logically in state M, this can also be considered the logical start state. The logical end state can potentially be determined by inspecting the function body for a `set` operation that transitions to a stable state, furthermore in PCC some transactions can be specified with a deterministic end state, which is then encoded into the transient state.

Given that we have determined the logical start state and potentially logical end state(s) for a transient state, how can we determine which forwarded messages could potentially arrive and if so how do we handle them? Once we have obtained the logical states we can inspect which handlers exist for which messages in the defined protocol and this will indicate that we can expect that message to arrive. For example, again consider the transaction in Figure 3.3 and the transient state `M_evict`. We know that this state has logical start state M and (from the PCC definition) logical end state I. Therefore, we have to look at each of these stable states and determine which forwarded messages each of these stable states handles, because they can potentially arrive while in the transient state `M_evict`. If a message arrives, that can be handled by the logical start state, we know that the directory received a request before the arrival of the initial message, causing the directory to forwarded the request. However if the forwarded request is one that is handled by the logical end state, then the directory received a request after our initial message.

Given that we have received a forwarded message and we have determined in which logical state the directory *sees* the cache in, which informs the cache and allows it to deduce the ordering of requests, how do we then handle this message? As mentioned before, we have two cases to consider: (1) request is handled by the logical start state, meaning that another request is ordered before and (2) the request is handled by the logical end state which means a a request was ordered after our initial request.

Case (1) *Handled by Logical Start State* : We have deduced that the forwarded message

Figure 3.4: S→M Transaction in MSI Protocol

relates to the the logical start state and that the directory received our request after. In this situation it is critical that the cache respond immediately to the request without delay, crucially the cache must not continue to wait for its original response because this could cause deadlock to arise. For example, suppose that two caches C1 & C2 both currently is State S, simultaneously initiate the S→M Transaction (see Figure 3.4), thereby issuing `GetM` requests to the directory. Suppose, that the directory breaks the tie by ordering C1's request before C2's, therefore it sends an invalidate to C2, allowing C2 to determine that its request is ordered after as this request is handled by its logical start state S. If C2 were to stall this request and continue to wait for the original message, then once the directory processes C2's `GetM` request it will forward it to C1, the current owner. However, C1 cannot fulfil this request, until C2 has responded to the invalidate, which causes a circular dependency and therefore a deadlock.

Given that C2 must respond to the forwarded message, what transient state must it transition to now? The problem is that the directory has transitioned state after the cache had sent the original request, therefore the logical start state might change. In general, the cache must cancel its original transaction and issue a new request to transition to M state, however most directory protocols like MSI, issue the same request in all stable states when requesting M state. Therefore, it is not necessary to cancel the original transaction, but continue with the transaction as if it was sent while in a different stable state. In this case: after C2 receives the invalidate it logically transitions to state I, therefore when the directory *sees* C2's `GetM` request it will proceed with the transaction as if C2 is is state I. Therefore, after C2 processes the invalidate it must transition to a stable state that reflects the fact that it has logical start state I and performed a store (issuing a `GetM`), therefore it will transition to `I_store`. If this transient state does not yet exist then we create it, but then how to we proceed with the transaction if we do not know how the directory will respond? In some cases the response from the directory changes depending on the stable state we begin from, therefore we need to inspect the directory controller and determine which response will be sent, and then proceed to handle this accordingly.

Case (2) *Handled by Logical End State* : In this scenario we have determined that

the forwarded message is handled by the logical end state, meaning that the request was ordered after our own. Because we know that this request was order after, this means that the directory has already responded to our request and the response is in flight, we have just not received it yet. Therefore, it is safe to continue to wait for the response to arrive, without risking deadlock. This is known as stalling, and is the most straightforward way of handing forwarded messages that are associated with the logical end state. Stalling however is not the most efficient method for handling this scenario as it reduces the degree of concurrency between the two racing transaction, but it can also block the network messages for other blocks.

Because the forwarded request is associated with the logical end state, we may not yet have the permissions to fulfil this request until we have completed our own transaction. For example: if we are in `I_load` in the transaction in Figure 3.2 and we receive a forwarded `Fwd_GetM`, we cannot yet fulfil this request until we have fully transitioned to state M. However, instead of stalling, we can transition to a suitable transient state that reflects the fact that we still need to fulfil this request after we received the response from the directory. Thus, in essence we stall the request, but we unblock the network and allow other messages through. Although this optimization is implemented in ProtoGen, it is not currently implemented in ProtoGen-MLIR due to time constraints for completing this project. Any requests associated with the logical end state are stalled, until the original response arrives. Therefore, in scenarios in which the end state cannot be determined, as discussed before, ProtoGen-MLIR will stall this request until it is able to be processed.

However, it is possible to be more optimistic and respond to forwarded messages without stalling or deffering the message. For example, consider the scenario in the I→S transaction in the MSI protocol. A cache performs a load by issuing a `GetS` request to the directory transitioning to state `I_load`. While in this transient state it receives a forwarded `Inv` message; clearly this message is associated with the logical end state S. However, instead of stalling we can immediately send the `Inv_Ack` to the required cache, because the current cache will become invalidated anyway. However, we still haven't received the `GetS_Ack` response from the directory, which we know must arrive. Therefore, we have to transition to a transient state with logical start state I and waiting for a `GetS_Ack` from the directory.

From this reasoning we implement these optimizations on the protocol by again leveraging the MLIR supporting infrastructure. Similarly, to the method of removing `await` statements from the IR and generate additional transient states (discussed in Section 3.4), this step also involved directly matching and modifying the IR in place. We therefore match on every transient state defined in the protocol after removing `await` statements, and apply the following steps:

1. We get the logical start state from the transient state name i.e. `M_evict` → `M`
2. We then discover what messages are handled by this logical start state i.e. `Fwd_GetM`, each of which is then considered individually in the remaining steps
3. We handle the message as if in the stable state, executing the rules defined in the protocol

4. If this handler causes a transition to a new state, then we have to a transition to a new transient state that reflects this change. i.e. when a `Fwd_GetM` arrives in state M, this causes a transition to state I. This then causes the transition `M_evict`→`I_evict`
5. If this transition is to a state that does not yet exists, then we create it
6. If we created a new state, then we need to inspect at the directory controller and determine which message will be send as the directory *sees* us in a new state. We can add the final transition from transient state to stable state, when this message arrives.
7. Add any newly created transients states to the list still to be considered.

3.6 The Murphi Dialect

The Murphi Dialect is the MLIR compilation target, from which we can then generate valid Murphi code. The Murphi Dialect is functionally simpler than the PCC Dialect and only defines a set of function that represent the cache and directory controllers. Listing 3.8 shows such a function in the Murphi Dialect. The Dialect includes operations, similar to ones in the PCC Dialect such as sending messages and updating the state, but has no guarantees on transaction atomicity, therefore there is no such constructs such as `await`. The Murphi dialect also contains no reference to logical states, as this information is no longer needed for verification, and used only for implementing optimizations.

Conversion between PCC and Murphi dialects is called *lowering*, and this is usually done as the final step after all optimizations have been applied. More complex MLIR pipelines may lower through multiple dialects before reaching the desired output target, possibly with additional optimization realised at each stage, however ProtoGen-MLIR has only one such step.

```
"murphi.function"(){
    %msg = "murphi.msg_constr"(){msgType="Request", parameters=["GetM", "ID", "directory.ID"]} : () -> i64
    "murphi.send"(%msg) {netId="req"} : (i64) -> ()
    "murphi.set"(){id="State", value="cache_I_load"} : () -> ()
    "murphi.return" () :  () -> ()
}){machine="cache", cur_state="cache_I", action="load"} : () -> ()
```

Listing 3.8: Murphi Dialect Function

Some operations can easily be lowered simply by converting between the equivalent operations in the respective dialect, like the `ConstantOp` in Listing 3.9. However, some operations such as messages require a more complex conversion. Murphi supports only a single message type, therefore message construction operations also require a type conversion between dialects. Also additional operations are inserted, for specific boilerplate constructs that must be introduced in the final Murphi code, such as for `scalarsets` as detailed in section 2.4.

```
/* PCC Dialect */
%0 = "pcc.constant"(){ value = 3} : () -> i64
```

```
/* Murphi Dialect */
%0 = "murphi.constant"(){ value = 3} : () -> i64
```

Listing 3.9: PCC to Murphi Constant Op Conversion

3.7 Murphi Backend

After the final step of the MLIR pipeline is complete we are left with a valid and optimized protocol specified through MLIR operations in the Murphi Dialect from which we are ready to generate valid Murphi code. The MLIR Operations themselves only define the behaviour or our cache coherence protocol and thus to generate valid Murphi code we must define additional constructs to produce the Murphi code which can be validated.

A Murphi program consists of four parts: (1) forward declarations of all types, global variables and constants; (2) any additional procedures or functions; (3) a set of rules which can be executed when a condition is satisfied and (4) any invariants that must be maintained for each new state explored. To generate a valid Murphi program to successfully validate the protocol we must define all the necessary types and constants, including cache and directory definitions as well as a general message type; we must specify the behaviour of the cache and directory controller through procedures; and finally define a set of rules which Murphi can execute, for example when a message appears on a network or a cache wisher to perform a load.

Defining Appropriate types. Most types from PCC can easily be converted to Murphi such as integer ranges and sets, due to their frequent use in cache coherence protocols they are well supported in Murphi. Networks of any kind (Ordered and Unordered) can also easily be implemented in Murphi through boiler-plate declarations. However, types for the cache can directory controllers vary depending on the protocol, especially with regards to the auxiliary state they store. Luckily type specifications from PCC are naturally translatable to Murphi, Listing 3.10 shows the cache definition for a cache controller in the MI protocol.

```
ENTRY_cache: record
  State: cache_state;
  Perm: Access;
  cl: ClValue;
end;
```

Listing 3.10: Murphi Cache Type Definition

More difficult however is managing messages in the simulation. Networks in Murphi support the sending of a single message type, however most protocols use messages of different types that include different fields: for example in the MI Protocol a `Resp` message includes a `cl` field, while for example an `Ack` message does not. To handle this case we introduce a global `Message` type which can be used to represent any message and is a union of all the potential fields a message can contain. The global message declaration for the MI protocol can be seen Listing 3.11, and includes the `cl` field. To correctly construct messages of a specific type we then include factory

functions to return an instance of such a message that includes the necessary fields; any unused fields are set as undefined (see Listing 3.12). This modification does not change the protocol in any way and we can safely make this change without affecting the verification.

```
Message: record
      adr     : Address;
      mtype   : MessageType;
      src     : Machines;
      dst     : Machines;
      cl      : ClValue;
end;
```

Listing 3.11: Murphi Message Type

```
function Ack(adr: Address; mtype: MessageType; src: Machines; dst: Machines) :
    Message;
var msg: Message;
begin
  msg.adr := adr;
  msg.mtype := mtype;
  msg.src := src;
  msg.dst := dst;
  msg.cl := undefined;
  return msg;
end;
```

Listing 3.12: Murphi Message Type

Defining Cache and Directory Controllers. Once we have generated all the necessary types and constants that will be used during the simulation, we can begin to create the cache and directory controllers. As detailed, cache and directory controllers are effectively FSMs and can therefore be implemented as a simple function which performs an action based on the current cache state and the message it has received. In Listing 3.13, we present how such a function is implemented: this function switches over the current cache state, and then over the input message type to call the correct message handler. The message handler will then execute the correct MLIR instructions generated from the compiler, however the MLIR Operations too have to converted to valid Murphi.

```
function Func_cache(inmsg:Message; m:OBJSET_cache) : boolean;
var msg: Message;
begin
    alias adr: inmsg.adr do
    alias cache_entry: i_cache[m].CL[adr] do
        /* Switch over the current cache State*/
        switch cache_entry.State
            case cache_M:
                /* Switch over the Message Received */
                switch inmsg.mtype
                    case Fwd_GetM:
                        /* Call the correct handler function */
                        handle_cache_M_Fwd_GetM(m, inmsg);

                    /* -- Additional Msg Case Statements -- */
            endswitch;

            /* -- Additional State Case Statements -- */
        endswitch;
    endalias;
    endalias;
return true;
end;
```

Listing 3.13: Murphi Cache Handler FSM

To execute the correct message handler we use a simple function naming scheme of `handle_<cur_state>_<msg_type>` as only one such handler will ever exist, this allows us to specify the directory and cache controllers quite easily for each state (including transient states) and every message type. Therefore, all that is then additionally required is to correctly complete all the necessary handler functions which are detailed by the MLIR operations from the compiler. Listing 3.14 shows the raw MLIR Operations for the handler and Listing 3.15 show the conversion to Murphi. Note that in the Murphi handler in Listing 3.15 must forward declare any SSA values from the MLIR Operations for a valid murphi compilation: i.e. `var msg: Message;` is forward declaring the SSA value for the `murphi.msg_constr` Operation. We can see that there is a fairly natural translation from MLIR Operations to Murphi, however this step in particular proved quite challenging. Specifically, we faced challenges in constructing correct Murphi statements with the parameters provided by the MLIR Operations. For example, consider the `msg_const` Op in Listing 3.14, which uses as a parameter `Fwd_GetM.src` to refer to the origin of the input message. This is challenging to convert to Murphi directly because, Murphi has no concept that the concrete type `Fwd_GetM` exists because we unified all message types. Also the message it is referencing is passed in as a parameter to the `handle_` function so we cannot refer to as `Fwd_GetM`. To solve this, we need to statically analyze if the parameter is referencing the input message, and if so rewrite it correctly. However, to completely solve this problem, we would ideally modify the IR to refer to these parameters directly through MLIR references, which would make generating Murphi much easier as we do not have to perform the static analysis step. However we decided to focus our efforts on other aspects of the compiler, and leave this improvement as future work.

```
"murphi.function"() ( {
    %6 = "murphi.msg_constr"() {msgType = "Resp", parameters = ["GetM_Ack_D", "ID", "Fwd_GetM.src", "cl"]}
    "murphi.send"(%6, %2) : (i64, i64) -> ()
    "murphi.set"() {id = "State", value = "cache_I"} : () -> ()
    "murphi.return"() : () -> ()
}) {action = "Fwd_GetM", cur_state = "cache_M", machine = "cache"} : () -> ()
```

Listing 3.14: Murphi Operations for handling Fwd_GetM in state M

```
procedure handle_cache_M_Fwd_GetM(inmsg:Message; m:OBJSET_cache);
var msg: Message;
begin
    alias adr: inmsg.adr do
    alias cache_entry: i_cache[m].CL[adr] do

        msg := Resp(adr, GetM_Ack_D, m, inmsg.src, cache_entry.cl);
        Send_resp(msg);
        cache_entry.State := cache_I;

    endalias;
    endalias;
end;
```

Listing 3.15: State M, Fwd_GetM Handler (Murphi)

Rulesets. Once we have have defined the cache and directory controllers and all additional boiler-plate helper functions we must define a *ruleset* that specifies what actions

Murphi can execute when running the simulation. Logically, a cache can initiate a transaction in any stable state, therefore we define a *ruleset* that allows any cache controller to issue `load`, `store` & `evict` instructions in any stable state (See Listing 3.16). Furthermore, we define a *ruleset* for each network to pass a message to the cache controller or directory controller when a message appears on the network.

```
ruleset m:OBJSET_cache do
    ruleset adr:Address do
        alias cle:i_cache[m].CL[adr] do

            /* If in state I cache can perform a load */
            rule "cache_I_load"
                cle.State = cache_I
                    ==>
                SEND_cache_I_load(adr, m);
            endrule;

            /* -- additional rules  */
        endalias;
    endruleset;
endruleset;
```

Listing 3.16: Cache Rulesets

Invariants. Finally we can specify invariants which must be maintained for every explored state by Murphi. Listing 3.17 shows the implementation of the *Write Serialization* invariant detailed in Section 2.1, which check that no two caches can be in M(odified) state. If at any point while Murphi is verifying the protocol would this invariant be breached, Murphi would return a failed verification.

```
invariant "Write Serialization"
    forall c1:OBJSET_cache do
        forall c2:OBJSET_cache do
            forall a:Address do
                ( c1 != c2
                & i_cache[c1].CL[a].State = cache_M )
                ->
                ( i_cache[c2].CL[a].State != cache_M )
            endforall
        endforall
    endforall;
```

Listing 3.17: Write Serialization Invariant

Compilation. The generated Murphi file is then compiled with the Murphi compiler, which generates C++ code. This code is then finally compiled with `g++` into an executable and run. Murphi will then output the number of states explored upon successful verification, otherwise it will present the trace that lead to the counterexample.

Chapter 4

Testing

To test ProtoGen-MLIR we used it to compile standard MOESI protocols namely (MI, MSI & MESI) defined in PCC. The generated Murphi files were then compiled and executed to validate the protocol. We tested both the unoptimized compilation (with atomic transactions) as well as the optimized versions. We successfully compiled all protocols without optimizations, and a fully successful optimized compilation with MI. With MSI we achieved a partial successful compilation with optimizations, but we failed to produce a verifiable protocol with MESI.

4.1 MI Protocol

The MI Protocol, contains only two states: M(odified) for reading and writing; and I(invalid) for evicted or non cached blocks (See Figure 4.1). We specified this protocol baseline protocol in PCC (See Appendix A), which was then compiled as a stable protocol with atomic transactions. This introduces the additional transient states required for intermediate steps of a transaction and the generated protocol is shows in Figure 4.2. This atomic protocol was then compiled to Murphi, and was successfully validated for both deadlock freedom and the *Write Serialization* invariant detailed in Listing 3.15.

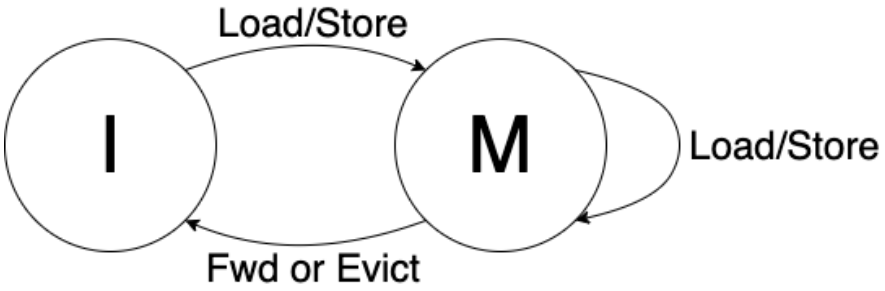

Figure 4.1: Caption

Upon successful compilation and verification of the stable protocol, we attempt to introduce additional concurrency through the methods detailed in Section 3.5. Figure

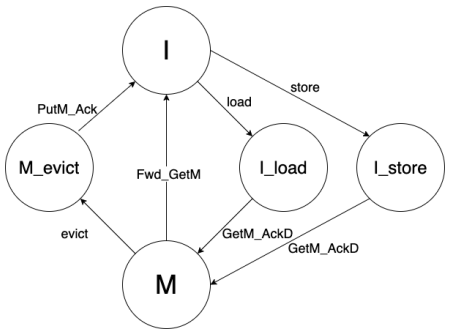

Figure 4.2: Caption

4.3 is the resulting protocol after compiling with optimizations enabled. It shows the addition of a transient state `I_evict` which was discovered. With the addition of this new transient state and the necessary transitions, we were able to remove the atomic transaction constraint and successfully validate the protocol for deadlock freedom and Write Serialization, without the use of locks.

This additional transient state was discovered by considering the state `M_evict`, and realising that a `Fwd_GetM` message can arrive, which is associated with the logical start state `M`. Knowing that after handling this request, the the cache will transition to state `I` (see 4.2), we must therefore transition to a suitable transient state to reflect the fact we are logically in state `I` and having performed an evict, hence `I_evict`. This transient state does not currently exist in the protocol, therefore ProtoGen-MLIR created it, adding the transition from the `M_evict` state. Finally, when ProtoGen-MLIR created this new state it realised that it was still going receive a `PutM_Ack` message from the directory, therefore the transition from `I_evict` to `I` is added.

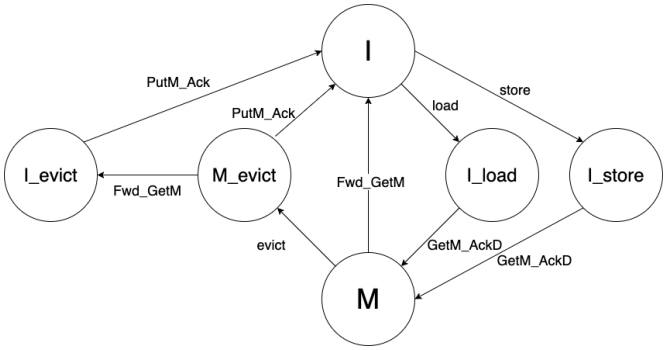

Figure 4.3: Optimized MI Protocol

4.2 MSI Protocol

The MSI protocol is a more optimal protocol than MI due the addition of the S(hared) state, allowing multiple caches to cache the same read-only copy of a block, thus utilising *Multiple Reader* from the SWMR invariant. Again we are able to verify the correctness this stable protocol using ProtoGen-MLIR, by compiling and executing the compiled Murphi file, and we were able to prove *Write Serialization* was maintained.

We provide the full protocol specified in PCC in Appendix B. The generated protocol, including all transient states can be seen in Figure 4.4. We also choose to present the directory controller in Figure 4.5. In the MI Protocol, the directory controller does not change because it has no transient states, due to its simplicity. However, in MSI when the directory receives a `GetS` from a cache while in state `M`, it needs to invalidate the current owner, and await until it receives a `WB` (WriteBack) to maintain coherence. This transaction in the directory introduces the additional transient state in the directory `M_GetS` as shown in Figure 4.5.

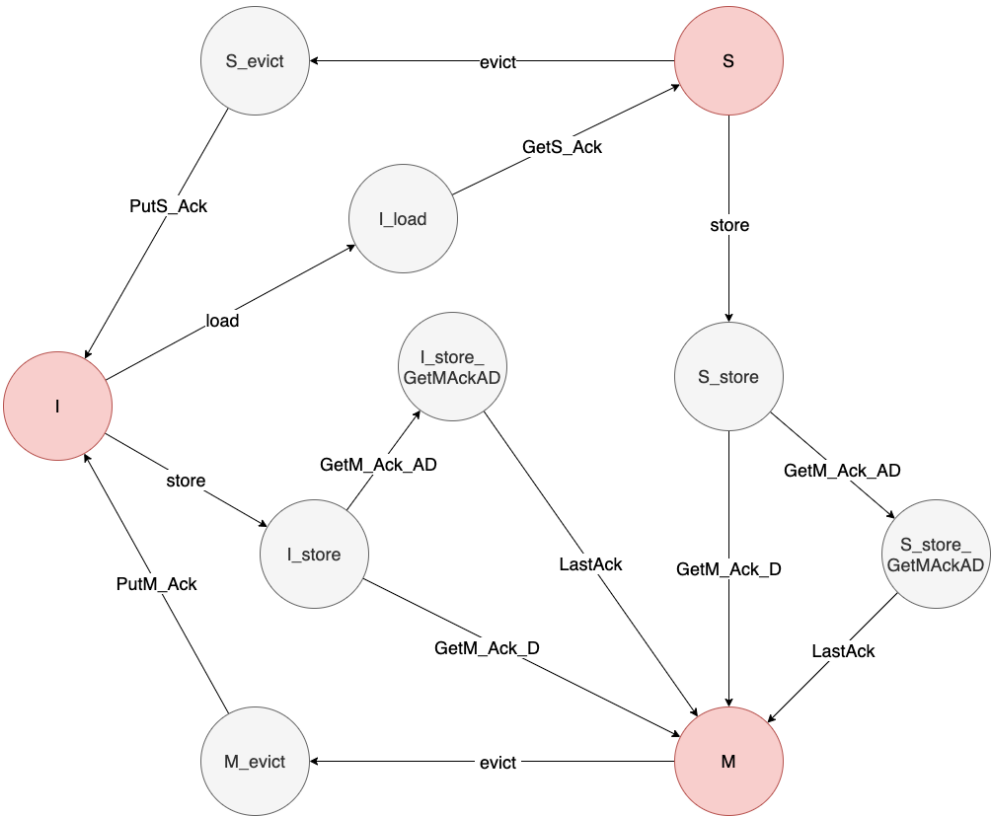

Figure 4.4: MSI Protocol (Atomic Transactions)

Compiling the MSI protocol with optimizations enabled yielded the FSM in Figure 4.6. Additional transient states are highlighted in green, with additional transitions emphasised in orange. Again we can see that ProtoGen-MLIR considers the `M_evict` state, realising that two forwarded messages can arrive for its logical start state, notably `Fwd_GetM` & `Fwd_GetS`, which need to transition to `I_evict` & `S_evict` respectively. The `I_evict` state did not exist, therefore it was created, similarly to MI. Similarly, ProtoGen-MLIR considered states `S_store` & `S_evict`, with logical start state S, and handled the possible forwarded `Inv` message accordingly. Although, this generated cache controller is correct, in the sense that it allows us to run the protocol without the use of locks, we were unable to verify it correctly in Murphi.

The reason for the failed verification is due to the fact, that the directory can transition state before a transaction message is received at the directory. Consider the case in MSI when a cache is in state S and wishes to transition to state M, with a S→M transaction, and issues a `Upgrade` message to the directory. However, suppose that

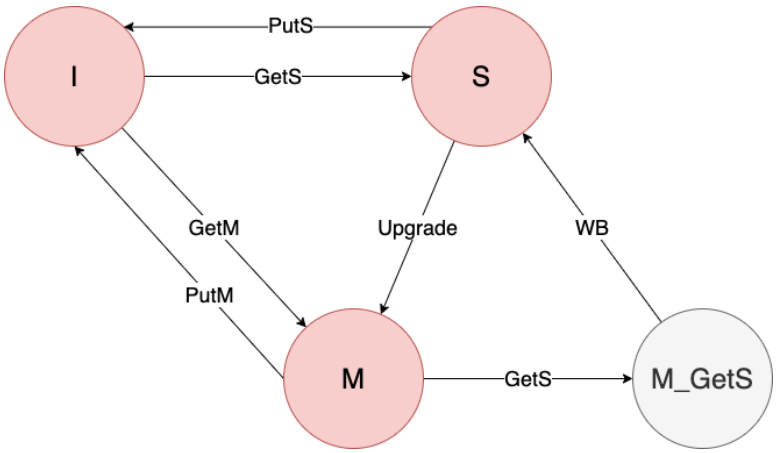

Figure 4.5: MSI Directory Controller

when in the transient state `S_load` an `Inv` message in received, causing the cache to transition to state `I_store` (see Figure 4.6). This was caused by another cache issuing an `Upgrade` request to the directory, causing it to send `Inv` messages to all sharers, and the directory is now in state M. However, when the initial `Upgrade` message is received at the directory there is no such handler, which causes the deadlock.

The key to overcoming this issue, is adding additional logic at the directory to realise that an `Upgrade` message can be re-interpreted as a `GetM` in state M and fulfil the request. This was realised too late on in development to change, but we present a solution as future work in section 5.2. However, manually updating the Murphi file to include this logic, yielded a verification without deadlocks, signalling that ProtoGen-MLIR is still functioning correctly.

4.3 MESI Protocol

MESI is a natural extension of the MSI protocol, and introduces the E(xclusive) state, which is issued to a cache on load requests, when the directory has no other sharers. Exclusive state allows a cache to perform write operations without again requesting permissions from the directory. Consider a process that is single threaded and therefore no sharers, this additional state allows the cache to *silently* upgrade its permissions without the delay of contacting the directory.

Once again we specify this protocol using PCC and the full definition can be seen in Appendix C. We can successfully compile and verify the protocol with atomic transactions enabled and produce the FSM detailed in Figure 4.7.

However, attempting to compile MESI with optimizations enabled did not yield a verifiable protocol. MESI suffers from the same problem discovered in MSI, but ProtoGen-MLIR also failed to find key transitions to prevent deadlock. For example, while considering the `E_evict` state, it failed to discover and handle the `Inv` message and transition to `I_evict`, which also was not created.

The reason that the protocol cannot be verifies is because the MESI protocol suffers

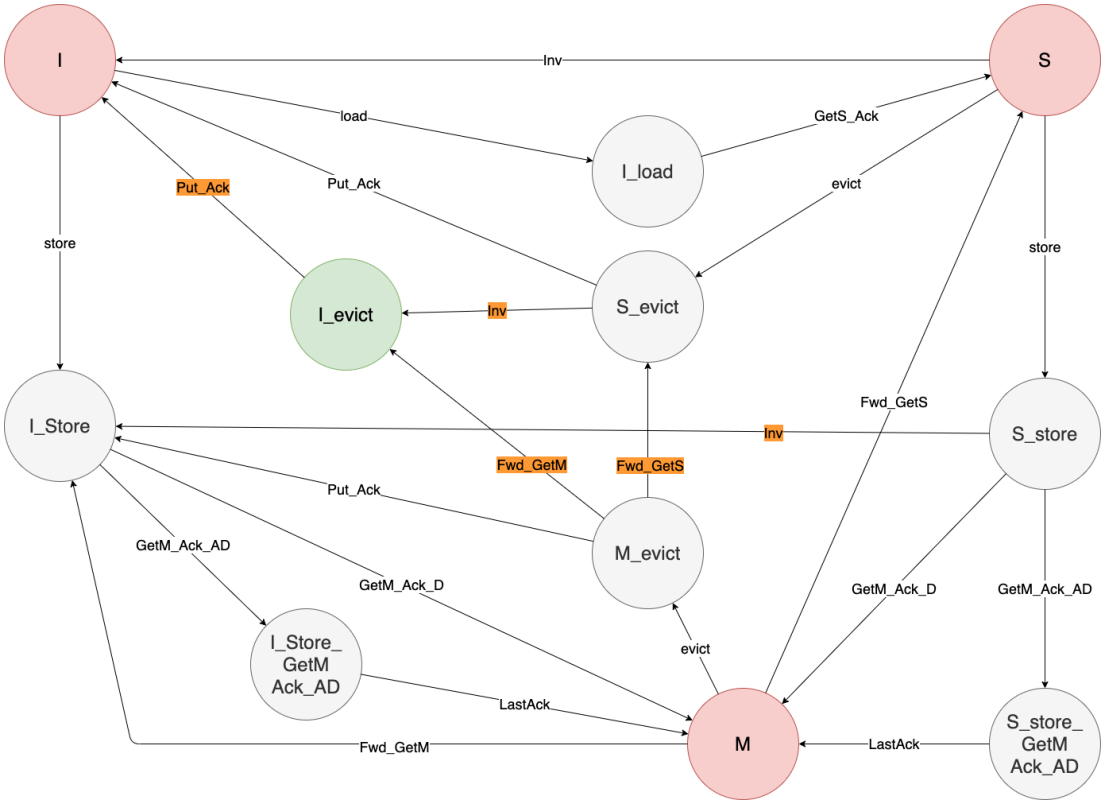

Figure 4.6: Optimized MSI Protocol

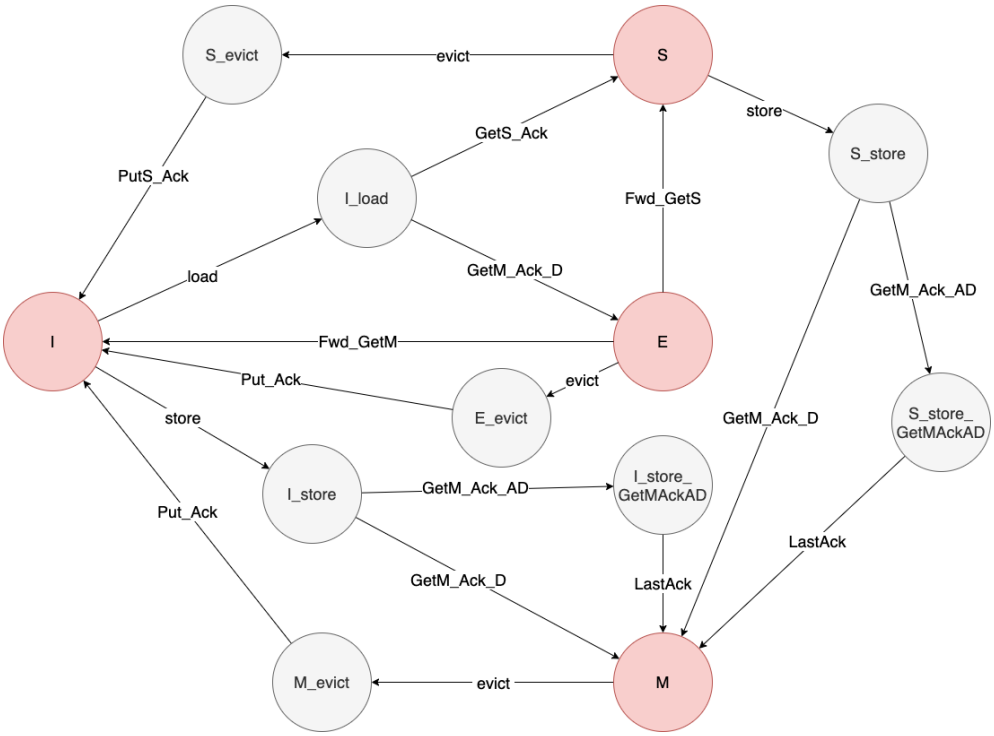

Figure 4.7: MESI Protocol

from another condition that is not handled in ProtoGen-MLIR. In MESI, both stable states M & E can receive the same forwarded message `Fwd_GetS`. This is a problem because consider a transient state that has logical start state E i.e. `E_evict`. ProtoGen-MLIR then assumes the message `Fwd_GetS` is associated with the logical start state E, however this message could be associated with the potential logical end state M. To solve this we need to pre-process the stable protocol and detect when these duplicate messages can arrive, and rename to break this ambiguity.

This pre-processing step is also required to aid with discovering branches in transactions to determine the logical end states. Therefore, it is a key step not only in being able to optimize the MESI protocol, but also to introduce additional concurrency in all protocols. With the limited time available it was difficult to fully implement this step, therefore we are not including it as part of this project, although work has been made towards it. However, in section 5.2 we present it as future work for part 2 of this project.

To summarise, these results are somewhat expected and align with the findings from ProtoGen. We also recognise how such errors are occurring, and understand that they do not breach our reasoning discussed in section 3.5

Chapter 5

Conclusions and Future Work

5.1 Summary

ProtoGen-MLIR is a novel approach to compiling stable state cache coherence protocols, specified in PCC. It leverages MLIR for its underlying compiler infrastructure, to implement the optimizations presented by ProtoGen [8]. ProtoGen-MLIR includes the following key contributions: a PCC frontend taht uses an ANTLR4 parser to construct an AST from which we can emit MLIR; an MLIR pipeline which modifies the IR to introduce transient states and optimizations and finally a Murphi backend implementation, which generates valid Murphi code for formal verification.

In section 2.1 we detailed the challenges of designing correct and efficient coherence protocols due to the large number of transient states and unexpected messages. We further detailed how coherence protocols can more naturally be reasoned about from stable definitions, and in section 3.1 we presented the PCC DSL, which provides a elegant method for specifying coherence protocols assuming atomic transactions.

Next in chapter 3 we detailed the implementation of the MLIR compiler. Firstly, in section 3.2, we present parsing PCC with ANTLR4 to generate an AST, which we traverse to emit un-optimized MLIR Operations in the PCC Dialect, detailed in section 3.3. In section 3.4, we detailed the process of decomposing transactions specified in PCC into transient states. We also present how additional locking operations must be inserted to allow such protocols to be verified formally with Murphi. In section 3.5 we detail the methodology and reasoning behind the optimizations to enabling us to lift the atomic transactions constraint and allow caches to synchronize without locks. We show how we can handle forwarded messages, by recognising their association with either logical start or end states, and take action accordingly, possibly by introducing additional transient states. In section 3.6 we present the final stage of the MLIR pipeline, by converting our protocol operations to a generic Murphi IR. This generic IR is then consumed by a Murphi backend implementation that can successfully generate valid Murphi code, which when compiled with the Murphi compiler, allows for formal verification.

Finally in chapter 4 we present our resulting for compiling the popular MOESI proto-

cols; MI, MSI & MESI. We successfully compiled all protocols without optimizations enabled, and produced a correct and verifiable Murphi output. Furthermore, we managed to successfully implement the optimizations for the MI protocol and partially for the MSI protocol, allowing us to lift the atomic transactions constraint, and verified these generated protocol for correctness with Murphi. Unfortunately, we failed to verify MESI with optimization enabled, but we detail the challenges which this protocol presents, and plans for how to proceed in part 2 of this project.

5.2 Plans for MInf Part 2

ProtoGen-MLIR is a robust implementation of the key components of the compilation pipeline. However, ProtoGen-MLIR does not come without its shortcomings. In this section we present some features which we seek to implement in part 2 of this project.

Optimized Compilation for full MOESI protocols. Although ProtoGen-MLIR can successfully compile stable version of all the MOESI protocols, it successfully applied the required optimizations to remove the atomic transaction constraints fully for only the MI protocol, and paritally for the MSI Protocol. This would involve the addition of the following: (1) the ability for the compiler to rename messages that arrive in the same stable state and (2) allowing the directory to reinterpret requests correctly for stale requests, therefore allowing successfully compilation of all MOESI protocols.

Respond to Forwarded Messages in the Logical End State. ProtoGen-MLIR does not support responding to forwarded requests that are associated with the logical end state, instead choosing to stall these requests until the cache has transitioned to the correct stable state. As mentioned stalling is highly inefficient due other messages becoming blocked on the network. Implementing the ability for non-stalling protocols, to allow the cache controller to consume the message and possibly optimistically respond to the message if it is able to. This would be a non-trivial task and would involve preprocessing the stable protocol to determine branches in all possible transactions before applying the optimizations, however the resulting protocols will be non-stalling.

Remove Equivalent States. Another potential for optimization is removing states from the optimized protocol that are equivalent. For example, consider the generated optimized MI protocol in Figure 4.3. Notice that we could potentially join states `I_load` and `I_store` into a single state. We can do this, because both accept the same message and transition to the same stable state. However, this stage must be performed as a final step, as state naming is key to generating an optimized protocol.

Support for Consistency-Directed Protocols. Consistency-agnostic protocols, are the set of protocols in which writes are propagated to other caches synchronously, meaning a write is visible to all caches before that transaction completes. This gives the illusion (to the processor) that they are interacting with an atomic memory system[7], as the coherence protocol effectively makes the caches invisible. The processor can perform loads and stores to satisfy its consistency model without coordinating with the coherence protocol. All the MOESI protocols naturally fit into this category, as for a processor write to occur the cache must obtain that block is state M, before fulfilling the request. Conversely, consistency-directed protocols, allow writes to be propagated

to other caches asynchronously, meaning a cache can allow a write to occur before it has become visible to all caches. This clearly could result in incoherence as potentially two different caches could hold different versions of the data. However, as long as we can ensure that the writes are propagated to the caches according to the consistency model, then we can still guarantee that a cache will *see* the most up-to-date data when it is read from or written to. Consistency-directed protocols are typically deployed in accelerators such as GPUs, which have seen a huge boom in recent years, therefore supporting such protocols is extremely important.

Bibliography

[1] S. Chandra, B. Richards, and J.R. Larus. Teapot: a domain-specific language for writing cache coherence protocols. *IEEE Transactions on Software Engineering*, 25(3):317–333, 1999.

[2] David L. Dill. The murϕ verification system. *Computer Aided Verification*, page 390–393, 1996.

[3] David L Dill and Ralph Melton. Murphi annotated reference manual, Jul 1996.

[4] Aarti Gupta, Malay K. Ganai, and Chao Wang. Sat-based verification methods and applications in hardware verification. *Formal Methods for Hardware Verification*, page 108–143, 2006.

[5] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law, 2020.

[6] Llvm. llvm/circt.

[7] Vijaya Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence (Second Edition)*. Morgan & Claypool Publishers, San Rafael, 2020.

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

[9] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[10] David A. Patterson. *Computer Organization and Design (5th edition)*. Elsevier, Amsterdam, 2014.

Appendix A

MI.pcc

```
# NrCaches 3

Network { Ordered fwd;     //FwdGetS, FwdGetM, Inv, PutAck
          Unordered resp; // Data, InvAck
          Unordered req;  //GetS, GetM, PutM
          };

Cache {
    State I;
    Data cl;
} set[NrCaches] cache;

Directory {
    State I;
    Data cl;
    ID owner;
} directory;

Message Request{};

Message Ack{};

Message Resp{
    Data cl;
};

Message RespAck{
    Data cl;
};

Architecture cache {

    Stable{I, M}

    // I ////////////////////////////////////////////////////
    Process(I, store, State){
        msg = Request(GetM, ID, directory.ID);
        req.send(msg);

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

        }
    }
```

```
    Process(I, load, State){
        msg = Request(GetM, ID, directory.ID);
        req.send(msg);

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


    // M ////////////////////////////////////////////////////
    Process(M, load, M){
    }

    Process(M, store, M){}

    Process(M, Fwd_GetM, I){
        msg = Resp(GetM_Ack_D, ID, Fwd_GetM.src, cl);
        resp.send(msg);
    }

    Process(M, evict, State){
        msg = Resp(PutM, ID, directory.ID, cl);
        req.send(msg);

        await{
            when Put_Ack:
                State = I;
                break;
        }
    }
}

Architecture directory {

    Stable{I, M}

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

    // M ////////////////////////////////////////////////////
    Process(M, GetM){
       msg = Request(Fwd_GetM, GetM.src, owner);
       fwd.send(msg);
       owner = GetM.src;
    }

    Process(M, PutM){
       msg = Ack(Put_Ack, ID, PutM.src);
       fwd.send(msg);

       if owner == PutM.src{
            cl = PutM.cl;
            State=I;
       }
    }

}
```

Appendix B

MSI.pcc

```
# NrCaches 3

Network { Ordered fwd;     //FwdGetS, FwdGetM, Inv, PutAck
          Unordered resp; // Data, InvAck
          Unordered req;   //GetS, GetM, PutM
          };

Cache {
    State I;
    Data cl;
    int[0..NrCaches] acksReceived = 0;
    int[0..NrCaches] acksExpected = 0;
} set[NrCaches] cache;

Directory {
    State I;
    Data cl;
    set[NrCaches] ID cache;
    ID owner;
} directory;

Message Request{};

Message Ack{};

Message Resp{
    Data cl;
};

Message RespAck{
    Data cl;
    int[0..NrCaches] acksExpected;
};

Architecture cache {

    Stable{I, S, M}

    // I ////////////////////////////////////////////////////
    Process(I, load, State){
        msg = Request(GetS, ID, directory.ID);
        req.send(msg);

        await{
            when GetS_Ack:
                cl=GetS_Ack.cl;
                State = S;
                break;
        }
```

```
    }

    Process(I, store, State){
        msg = Request(GetM, ID, directory.ID);
        req.send(msg);
        acksReceived = 0;

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

            when GetM_Ack_AD:
                acksExpected = GetM_Ack_AD.acksExpected;

                if acksExpected == acksReceived{
                    State = M;
                    break;
                }

                await{
                    when Inv_Ack:
                        acksReceived = acksReceived + 1;

                        if acksExpected == acksReceived{
                            State = M;
                            break;
                        }
                    }

            when Inv_Ack:
                acksReceived = acksReceived + 1;
        }
    }

    // S ////////////////////////////////////////////////////
    Process(S, load, S){}

    Process(S, store, State){
        msg = Request(Upgrade, ID, directory.ID);
        req.send(msg);
        acksReceived = 0;

        await{
            when GetM_Ack_D:
                State = M;
                break;

            when GetM_Ack_AD:
                acksExpected = GetM_Ack_AD.acksExpected;

                if acksExpected == acksReceived{
                    State = M;
                    break;
                }

                await{
                    when Inv_Ack:
                        acksReceived = acksReceived + 1;

                        if acksExpected == acksReceived{
                            State = M;
                            break;
                        }
                    }

            when Inv_Ack:
                acksReceived = acksReceived + 1;
        }
```

```
    }

    Process(S, evict, State){
        msg = Request(PutS, ID, directory.ID);
        req.send(msg);

        await{
            when Put_Ack:
                State = I;
                break;
        }
    }

    Process(S, Inv, I){
        msg = Resp(Inv_Ack, ID, Inv.src, cl);
        resp.send(msg);
    }


    // M ////////////////////////////////////////////////////
    Process(M, load){
    }

    Process(M, store, M){}

    Process(M, Fwd_GetM, I){
        msg = Resp(GetM_Ack_D, ID, Fwd_GetM.src, cl);
        resp.send(msg);
    }

    Process(M, Fwd_GetS, S){
        msg = Resp(GetS_Ack, ID, Fwd_GetS.src, cl);
        resp.send(msg);
        msg = Resp(WB, ID, directory.ID, cl);
        resp.send(msg);
    }

    Process(M, evict, State){
        msg = Resp(PutM, ID, directory.ID, cl);
        req.send(msg);

        await{
            when Put_Ack:
                State = I;
                break;
        }
    }

}

Architecture directory {

    Stable{I, S, M}

    // I ////////////////////////////////////////////////////
    Process(I, GetS, S){
       cache.add(GetS.src);
       msg = Resp(GetS_Ack, ID, GetS.src, cl);
       resp.send(msg);
    }

    Process(I, GetM, M){
        msg = RespAck(GetM_Ack_AD, ID, GetM.src, cl, cache.count());
        resp.send(msg);
        owner = GetM.src;
    }

    // S ////////////////////////////////////////////////////
    Process(S, GetS){
       cache.add(GetS.src);
```

```
        msg = Resp(GetS_Ack, ID, GetS.src, cl);
        resp.send(msg);
    }

    Process(S, Upgrade){
        if cache.contains(Upgrade.src){
            cache.del(Upgrade.src);
            msg = RespAck(GetM_Ack_AD, ID, Upgrade.src, cl, cache.count());
            resp.send(msg);
            State=M;
            break;
        } else {
            msg = RespAck(GetM_Ack_AD, ID, Upgrade.src, cl, cache.count());
            resp.send(msg);
            State=M;
            break;
        }
        msg = Ack(Inv, Upgrade.src, Upgrade.src);
        fwd.mcast(msg, cache);
        owner = Upgrade.src;
        cache.clear();
    }

    Process(S, PutS){
        msg = Resp(Put_Ack, ID, PutS.src, cl);
        fwd.send(msg);
        cache.del(PutS.src);

        if cache.count() == 0{
            State=I;
            break;
        }
    }

    // M ////////////////////////////////////////////////////
    Process(M, GetS){
        msg = Request(Fwd_GetS, GetS.src, owner);
        fwd.send(msg);
        cache.add(GetS.src);
        cache.add(owner);

        await{
            when WB:
                if WB.src == owner{
                    cl = WB.cl;
                    State=S;
                }
        }

    }

    Process(M, GetM){
        msg = Request(Fwd_GetM, GetM.src, owner);
        fwd.send(msg);
        owner = GetM.src;
    }

    Process(M, PutM){
        msg = Ack(Put_Ack, ID, PutM.src);
        fwd.send(msg);
        cache.del(PutM.src);

        if owner == PutM.src{
            cl = PutM.cl;
            State=I;
        }
    }

}
```

Appendix C

MESI.pcc

```
# NrCaches 3

Network { Ordered fwd;     //FwdGetS, FwdGetM, Inv, PutAck
          Unordered resp; // Data, InvAck
          Unordered req;   //GetS, GetM, PutM
          };

Cache {
    State I;
    Data cl;
    int[0..NrCaches] acksReceived = 0;
    int[0..NrCaches] acksExpected = 0;
} set[NrCaches] cache;

Directory {
    State I;
    Data cl;
    set[NrCaches] ID cache;
    ID owner;
} directory;

Message Request{};

Message Ack{};

Message Resp{
    Data cl;
};

Message RespAck{
    Data cl;
    int[0..NrCaches] acksExpected;
};

Architecture cache {

    Stable{I, S, E, M}

    // I ////////////////////////////////////////////////////
    Process(I, load, State){
        msg = Request(GetS, ID, directory.ID);
        req.send(msg);

        await{
            when GetS_Ack:
                cl=GetS_Ack.cl;
                State = S;
                break;
```

```
            when GetM_Ack_D:
                cl=GetM_Ack_D.cl;
                State = E;
                break;
        }
    }

    Process(I, store, State){
        msg = Request(GetM, ID, directory.ID);
        req.send(msg);
        acksReceived = 0;

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

            when GetM_Ack_AD:
                acksExpected = GetM_Ack_AD.acksExpected;

                if acksExpected == acksReceived{
                    State = M;
                    break;
                }

                await{
                    when Inv_Ack:
                        acksReceived = acksReceived + 1;

                        if acksExpected == acksReceived{
                            State = M;
                            break;
                        }
                    }

            when Inv_Ack:
                acksReceived = acksReceived + 1;
        }
    }

    // S ////////////////////////////////////////////////////
    Process(S, load){}

    Process(S, store, State){
        msg = Request(GetM, ID, directory.ID);
        req.send(msg);
        acksReceived = 0;

        await{
            when GetM_Ack_D:
                State = M;
                break;

            when GetM_Ack_AD:
                acksExpected = GetM_Ack_AD.acksExpected;

                if acksExpected == acksReceived{
                    State = M;
                    break;
                }

                await{
                    when Inv_Ack:
                        acksReceived = acksReceived + 1;

                        if acksExpected == acksReceived{
                            State = M;
                            break;
                        }
```

```
                }

            when Inv_Ack:
                acksReceived = acksReceived + 1;
        }
    }

    Process(S, evict, State){
        msg = Request(PutS, ID, directory.ID);
        req.send(msg);

        await{
            when Put_Ack:
                State = I;
                break;
        }
    }

    Process(S, Inv, I){
        msg = Resp(Inv_Ack, ID, Inv.src, cl);
        resp.send(msg);
    }


    // M ////////////////////////////////////////////////////////
    Process(M, load){
    }

    Process(M, store){}

    Process(M, Fwd_GetM, I){
        msg = Resp(GetM_Ack_D, ID, Fwd_GetM.src, cl);
        resp.send(msg);
    }

    Process(M, Fwd_GetS, S){
        msg = Resp(GetS_Ack, ID, Fwd_GetS.src, cl);
        resp.send(msg);
        msg = Resp(WB, ID, directory.ID, cl);
        resp.send(msg);
    }

    Process(M, evict, State){
        msg = Resp(PutM, ID, directory.ID, cl);
        req.send(msg);

        await{
            when Put_Ack:
                State = I;
                break;
        }
    }

    // E ////////////////////////////////////////////////////////
    Process(E, load){
    }

    Process(E, store, M){}

    Process(E, Fwd_GetM, I){
        msg = Resp(GetM_Ack_D, ID, Fwd_GetM.src, cl);
        resp.send(msg);
    }

    Process(E, Fwd_GetS, S){
        msg = Resp(GetS_Ack, ID, Fwd_GetS.src, cl);
        resp.send(msg);
        msg = Resp(WB, ID, directory.ID, cl);
        resp.send(msg);
    }
```

```
    Process(E, evict, State){
        msg = Ack(PutE, ID, directory.ID);
        req.send(msg);

        await{
            when Put_Ack:
                State = I;
                break;
        }
    }

}

Architecture directory {

    Stable{I, S, E, M}

    // I ////////////////////////////////////////////////////
    Process(I, GetS, E){
       msg = Resp(GetM_Ack_D, ID, GetS.src, cl);
       resp.send(msg);
       owner = GetS.src;
    }

    Process(I, GetM, M){
        msg = RespAck(GetM_Ack_AD, ID, GetM.src, cl, cache.count());
        resp.send(msg);
        owner = GetM.src;
    }

    // S ////////////////////////////////////////////////////
    Process(S, GetS){
       msg = Resp(GetS_Ack, ID, GetS.src, cl);
       resp.send(msg);
       cache.add(GetS.src);
    }

    Process(S, GetM){
       if cache.contains(GetM.src){
           cache.del(GetM.src);
           msg = RespAck(GetM_Ack_AD, ID, GetM.src, cl, cache.count());
           resp.send(msg);
           State=M;
       } else {
           msg = RespAck(GetM_Ack_AD, ID, GetM.src, cl, cache.count());
           resp.send(msg);
           State=M;
       }
       msg = Ack(Inv, GetM.src, GetM.src);
       fwd.mcast(msg, cache);
       owner = GetM.src;
       cache.clear();
    }

    Process(S, PutS){
       msg = Resp(Put_Ack, ID, PutS.src, cl);
       fwd.send(msg);
       cache.del(PutS.src);

       if cache.count() == 0{
            State=I;
            break;
       }
    }

    // M ////////////////////////////////////////////////////
    Process(M, GetS){
       msg = Request(Fwd_GetS, GetS.src, owner);
       fwd.send(msg);
```

```
        cache.add(GetS.src);
        cache.add(owner);

        await{
            when WB:
                if WB.src == owner{
                    cl = WB.cl;
                    State = S;
                }
        }

    }

    Process(M, GetM){
        msg = Request(Fwd_GetM, GetM.src, owner);
        fwd.send(msg);
        owner = GetM.src;
    }

    Process(M, PutM){
        msg = Ack(Put_Ack, ID, PutM.src);
        fwd.send(msg);
        cache.del(PutM.src);

        if owner == PutM.src{
            cl = PutM.cl;
            State=I;
        }
    }

    // E ////////////////////////////////////////////////////
    Process(E, GetS){
        msg = Request(Fwd_GetS, GetS.src, owner);
        fwd.send(msg);
        cache.add(GetS.src);
        cache.add(owner);

        await{
            when WB:
                if WB.src == owner{
                    cl = WB.cl;
                    State=S;
                }
        }

    }

    Process(E, GetM){
        msg = Request(Fwd_GetM, GetM.src, owner);
        fwd.send(msg);
        owner = GetM.src;
        State=M;
    }

    Process(E, PutE){
        msg = Ack(Put_Ack, ID, PutE.src);
        fwd.send(msg);
        cache.del(PutE.src);

        if owner == PutE.src{
            State=I;
        }
    }

}
```